“十五”国家重点电子出版物规划项目　计算机知识普及和软件开发系列
精通热门软件工具丛书（4）



北京希望电子出版社　总策划
浦滨　编著

C游戏编程
从入门到精通

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

责任编辑　伯明天
封面设计　张志



本版书特点

102个C游戏编程实例
知识与编程实作相结合
具体、实用、可操作性强

适合对象

初、中级用户
大学和专业学校师生
专业游戏开发程序员和业余爱好者



CX-83657
定价：46.00 元
ISBN 7900088458



CX-3680
定价:35.00 元
ISBN 7900088490



CX-3686
定价:42.00 元
ISBN 7900088717



CX-3747
定价:39.00 元
ISBN 7900101012

ISBN 7-900101-01-2
9 787900 101013 >

ISBN 7-900101-01-2
定价：39.00 元（本版CD）

"十五"国家重点电子出版物规划项目　计算机知识普及和软件开发系列
精通热门软件工具丛书（4）

北京希望电子出版社　总策划
浦滨　编著

C游戏编程从入门到精通

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

内 容 简 介

本书以C语言游戏编程入手，以**102**个实例，近**200**个函数较为系统地介绍了C基于游戏编程与开发的方法与技巧，内容丰富并相互包容，相互渗透。以实际的基于不同平台的游戏制作为背景，知识阐述与实际案例结合，深入浅出，具体、直观、全面，可操作性强；是一本难得的集入门、深入到精通C游戏编程的综合书籍。

该书将大学计算机及大量相关专业课程的知识运用到C语言游戏编程的实践中，并从C语言游戏编程实践角度来诠释计算机及许多其他专业课题，内容涉及计算机硬件、软件等方面的编程技术。具体内容包括，图形绘制、中文显示、动画实现、文件调用、内存使用、声卡调用、中断、内存驻留技术、接口技术、数据库实现、简单病毒、界面技术等进行了详尽的介绍，并且配以大量的源程序以及程序分析对所涉及的理论进行充分的讲解和支撑。

本书根据大学C语言教学需要，适合于高校计算机和数学相关专业的学生以及所有C语言爱好者。此外，对于C/C++语言的初、中级用户，业余爱好者学习与培训，以及有一定软件开发经验的程序员和专业技术人员也有很好的借鉴和参考价值。

本版CD内容为游戏实例的源代码、编译程序、游戏函数及数据库等。

系 列 盘 书：“十·五”国家重点电子出版物规划项目　计算机知识普及和软件开发系列
精通热门软件工具丛书（4）

盘 书 名：C游戏编程从入门到精通

总 策 划：北京希望电子出版社

文本著作者：浦　滨

CD制作者：希望多媒体开发中心

CD测试者：希望多媒体测试部

责任编辑：但明天

出版、发行者：北京希望电子出版社

地　　址：北京中关村大街26号，100080
网址：www.bhp.com.cn　E-mail: lwm@bhp.com.cn
电话：010-62562329, 62541992, 62637101, 62637102, 62633308, 62633309
（图书发行和技术支持）
010-62613322-215（门市）　010-62547735（编辑部）

经　　销：各地新华书店、软件连锁店

排　　版：希望图书输出中心　邓蛟龙

CD生产者：北京中新联光盘有限责任公司

文本印刷者：北京媛明印刷厂

开本/规格：787毫米×1092毫米　16开本　25.5印张　591千字

版次/印次：2002年5月第1版　2002年5月第1次印刷

本 版 号：ISBN 7-900101-01-2

印　　数：0001-4000

定　　价：39.00元（本版CD）

说明：凡我社产品如有残缺，可执相关凭证与本社调换。

前　言

本书从 C 语言游戏编程入手，对图形绘制、中文显示、动画实现、文件调用、内存使用、声卡调用、中断、内存驻留技术、接口技术、数据库实现、简单病毒、界面技术等进行了详尽的介绍，并且配以大量的源程序以及程序分析对所涉及的理论进行充分的讲解和支撑。

该书将大学计算机及相关专业大量课程的知识运用到 C 语言游戏编程的实践中。这些课程包括：C 语言程序设计、C++语言程序设计、汇编语言、高等数学、计算机硬件、接口技术、数据库原理、数据结构、算法设计、软件工程。此外，还涉及到图形技术、动画技术、病毒与病毒防治编程等一系列专业知识。

本书共包括 17 章，具体内容分别为：

第 1 章 猜数字游戏：通过一个简单的游戏引导大家认识游戏编程。

第 2 章 语言函数库画图：介绍 C 语言自身提供的绘图函数。

第 3 章 简单动画：利用 C 语言绘图函数制作简单的动画效果。

第 4 章 简单图形游戏：在动画的基础上实现简单的图形游戏。

第 5 章 图形模式：介绍在制作游戏前如何将屏幕从文本模式设置为图形模式。

第 6 章 二维图形：介绍如何自行设计基本图形函数。

第 7 章 中文显示：主要介绍在图形中如何实现中文。

第 8 章 图形文件：图形文件的调用和显示，介绍 bmp、pcx 和 ico 文件的算法和显示。

第 9 章 动画原理：全面介绍各种动画实现的方法，并给出例程。

第 10 章 子画面技术：介绍最流行的子画面动画技术。

第 11 章 文件操作：文件的建立、写入和读取，主要使用于游戏进度保存和游戏数据读取。

第 12 章 声音技术：主要介绍扬声、声卡的使用，以及在游戏中的实现背景音乐。

第 13 章 内存技术：主要介绍 xms、EMS 内存技术及其在游戏中的应用。

第 14 章 接口技术：主要介绍键盘、鼠标和串口的驱动和使用方法。

第 15 章 界面技术：为游戏提供具有面向对象思路的图形、操作界面。

第 16 章 其它问题：通过加密和病毒保证游戏的版权。

第 17 章 游戏例程：通过一个实际的游戏编写来整合前面各章节学到的内容。

从构建一个全方位的 C 语言游戏实用函数库，使得函数数量达到 200 个左右。函数类别包括：图形函数、动画函数、图形文件调用函数、数据文件处理函数、中文显示函数、内存函数、声音函数、计算机基本功能设置函数、接口函数、界面函数、数据库函数等。需要说明的是，本游戏函数库是在对 Andre LaMothe 等编写的运行于 Microsoft C7.0 平台下的函数库进行平台改写、函数大量修改和扩充以及增加了 100 多个其他方面函数的基础上实现的。此外，本书还包括 108 个例程。

内容全面。游戏编程全方位阐述；涉及计算机各相关专业课程；适于编写各类型游戏，包括桌面智力游戏、视频战斗游戏、简单 RPG 游戏、简单联机游戏等。

在创新方面，将 TC 作为游戏编程平台。保护模式（TC 没有）等更适合写游戏，所以很少有人用 TC 写游戏；

其次，从游戏角度学习 C 语言，从一个新的角度看待编程教学。教师通常从专家角度进行教学，于是在基础教学过程中往往和学生脱节（学生无法理解其很多表达含义）。本书的内容和程序有不少是从大一到大四过程中的笔记修改而来的，与读者学习进度非常接近，从而更容易吸收。

在过程性语言中更多加入面向对象思路。在游戏对象结构定义的时候融入更多属性、事件函数指针和链表指针，在子画面处理和界面对象处理中更多采用了事件驱动的思路，从而为今后转向 C++游戏编程铺平道路。

此外，将计算机相关专业知识通过 C 语言游戏编程的视角进行了解、学习或巩固。平时学习各科目内容较为独立或者仅仅基于理论上的联系，通过游戏编程我们可以将它们贯穿起来学习，触类旁通。

本书读者对象：

大学计算机、数学及相关专业学生；游戏爱好者。

另外，本书适合大学 C 语言教学需要，内容衔接和难度上由浅入深，着重衔接 C 语言基础教材；也适合于高校计算机和数学相关专业的学生以及所有 C 语言爱好者。

游戏编程其实就是编程！如果你接触过程序设计，那么你打算写一个游戏已经不再是什么困难的事情。如果你认为自己虽然学过 C 语言，然而对于编程只有一点初步的了解，根本谈不上编写游戏的问题。我们是否有编游戏的基础呢？不过相信在看这本书的时候，你一定具备了：

(1) 编程的初步基础；

(2) 对编写游戏的强烈欲望。

1. 游戏编程的几大要素

一个成功的游戏通常有如下价值和特点。

（1）极高的美工水平；

（2）逼真的动画效果；

（3）精巧的构思；

（4）简便的操作。

可以说这与编程的关系不算很大，只有部分动画和用户操作是需要在编程中下苦功夫的；而游戏成功的更重要原因是构思的独特和美工制作的赏心悦目。

以下是根据游戏的要素和编程的特点提出的游戏编程的五大要素：

（1）动画效果。动画的效果一定要流畅，没有散动和尽量逼真，最好能够符合“物理学”和“生物学”运动规律。

（2）人机交互性。游戏易于操作，并且能够快速响应，必要的时候能够设计出适合游戏的输入设备和驱动程序。

（3）媒体多样性。提高图像、动画、声音和操作的同步性和混合性。

（4）数据结构高效性。用于实现游戏的数据结构一定要经过规划，尽量简便、高效，以及适用面和扩充性强。

（5）代码的通用性和对象化。引入面向对象的思路将对理清程序员思路、简化程序设计以及程序扩充等问题带来相当大的帮助。

这些编程要素事实上涉及到了包括直接写屏、中断、多任务、内存技术、动画技术、显示技术和优化算法等各种编程技术；涉及到了内存、PC 喇叭、声卡、键盘、鼠标、手柄、显卡等各种硬件原理和硬件编程；也涉及了包括《C 语言程序设计》、《C++语言程序设计》、《汇编语言》、《数据结构》、《数据库》、《计算机硬件》、《接口技术》、《算法》、《高等数学》、《概率论》、《人工智能》、《软件工程》和《多媒体技术》等计算机专业课目；此外，还涉及到《动画技术》、《三维游戏编程》、《加密、解密技术》和《病毒与病毒防治编程》等一些当前流行的计算机专业技术。

具备良好的计算机专业知识是实现游戏编程各大要素的前提。当然也不用太担心许多专业知识你还不太懂。因为这里毕竟是游戏编程，编程能力才是真正的关键，一切运用到的相关专业知识也都是围绕游戏编程的，在之后的章节中会根据相应编程的需求有所阐述。

其实，游戏编程可以看作动画编程、输入输出设备编程和多媒体编程的集合，自然在写程序前应当设计好数据结构及算法。

2. 游戏的流程

别看游戏有那么复杂的情节和玩法，从开始到发展再到结束的流程中，归根结底就只有四样东西：

动画　根据变量运算结果对屏幕中变化的对象进行重画，从而实现动画效果；

响应　对于来自键盘、鼠标等输入设备和计算机内部如时间等中断进行响应；

运算　根据各类具体响应通过计算和重新赋值来改变程序内变量数值；

循环　反复进行动画、响应和运算的操作来实现游戏的真正进程。

实现一个游戏的基本流程如右（流程图 A）。

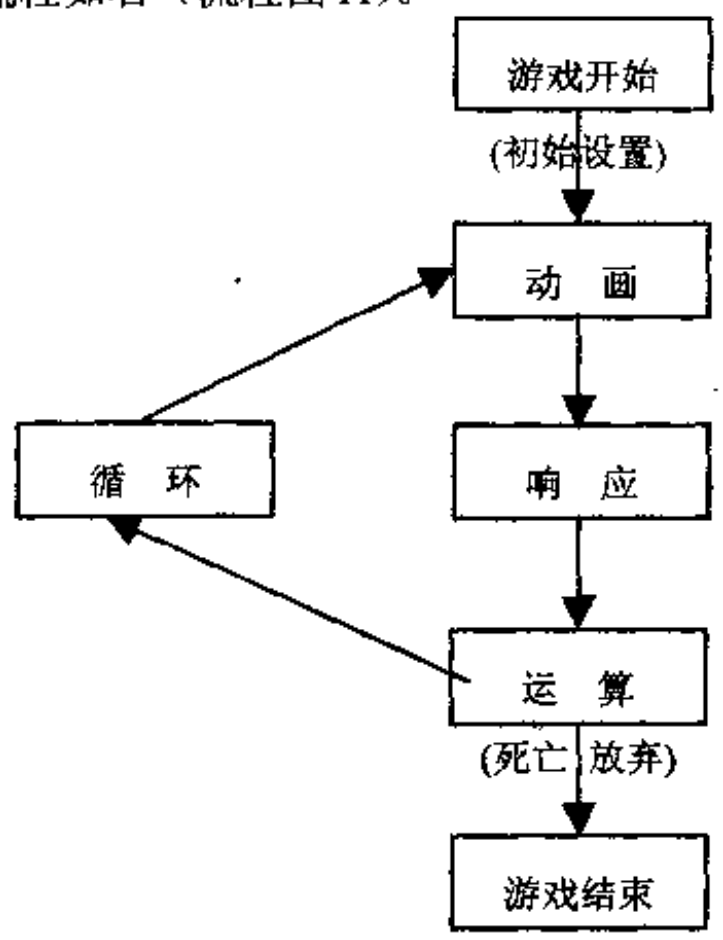


图 A　流程示意

下面以俄罗斯方块为例，动画流程分为以下几个步骤（图 B、图 C、图 D、图 E 及图 F）：



图 B



图 C

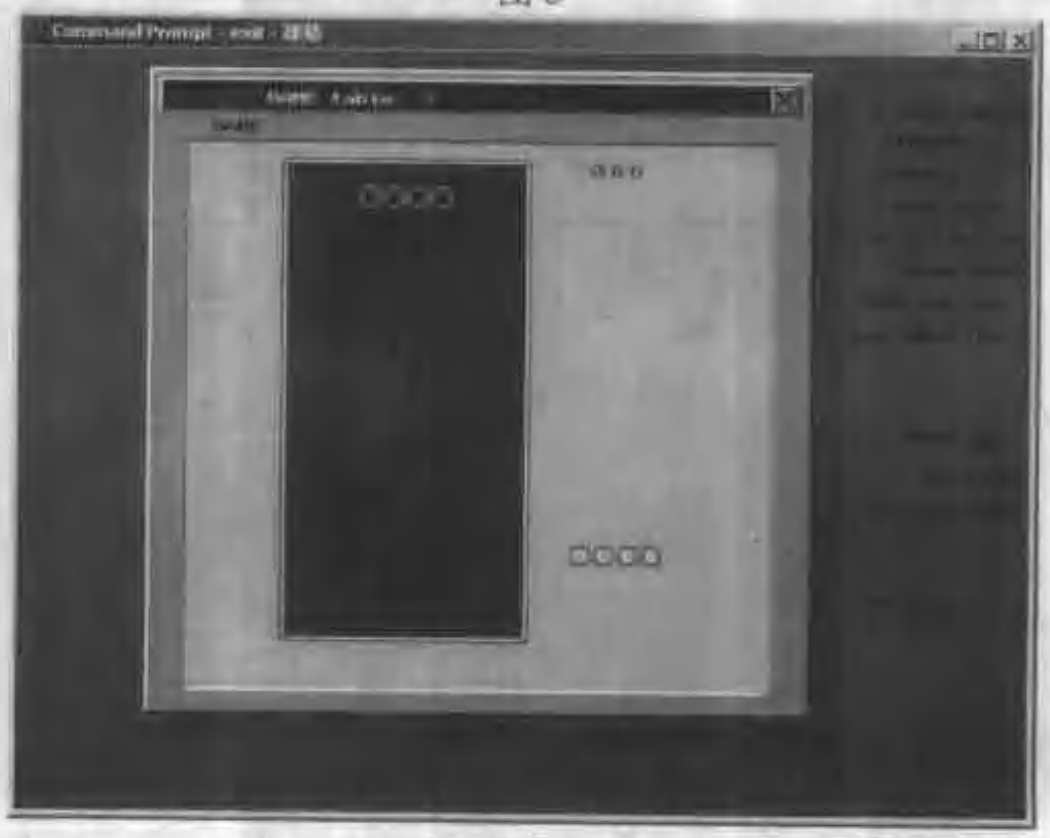


图 D

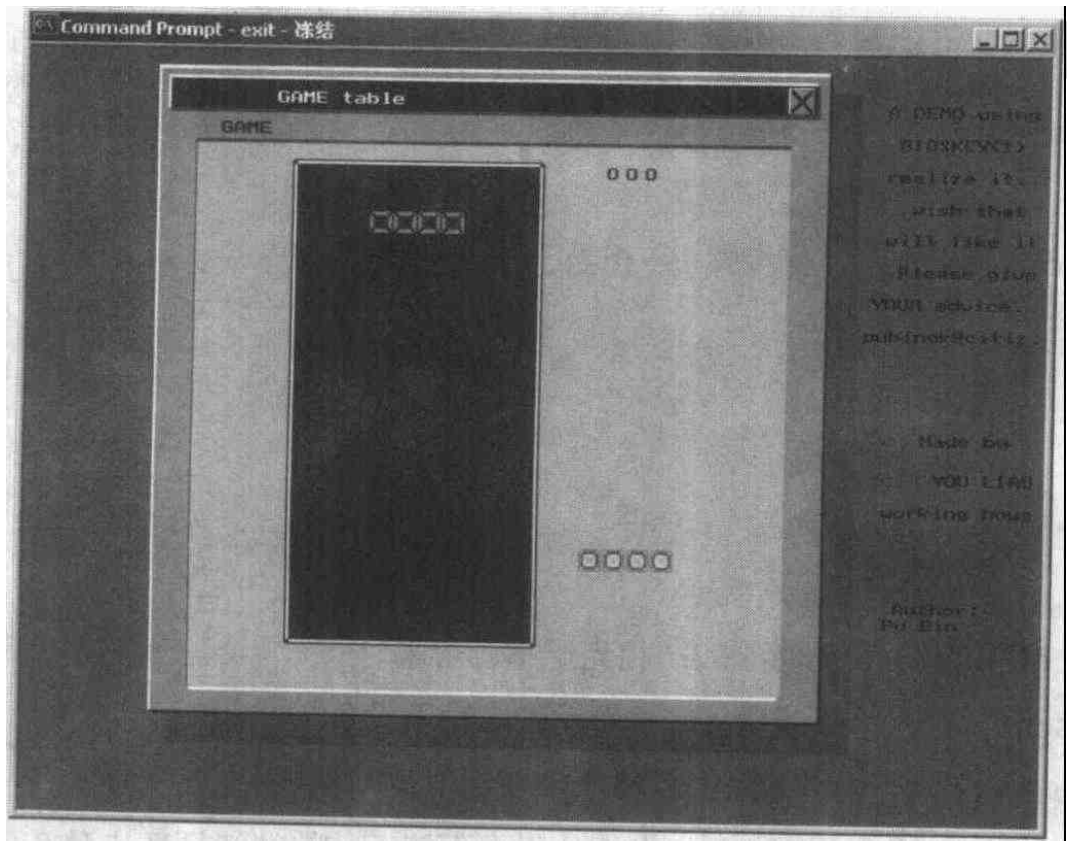


图 E

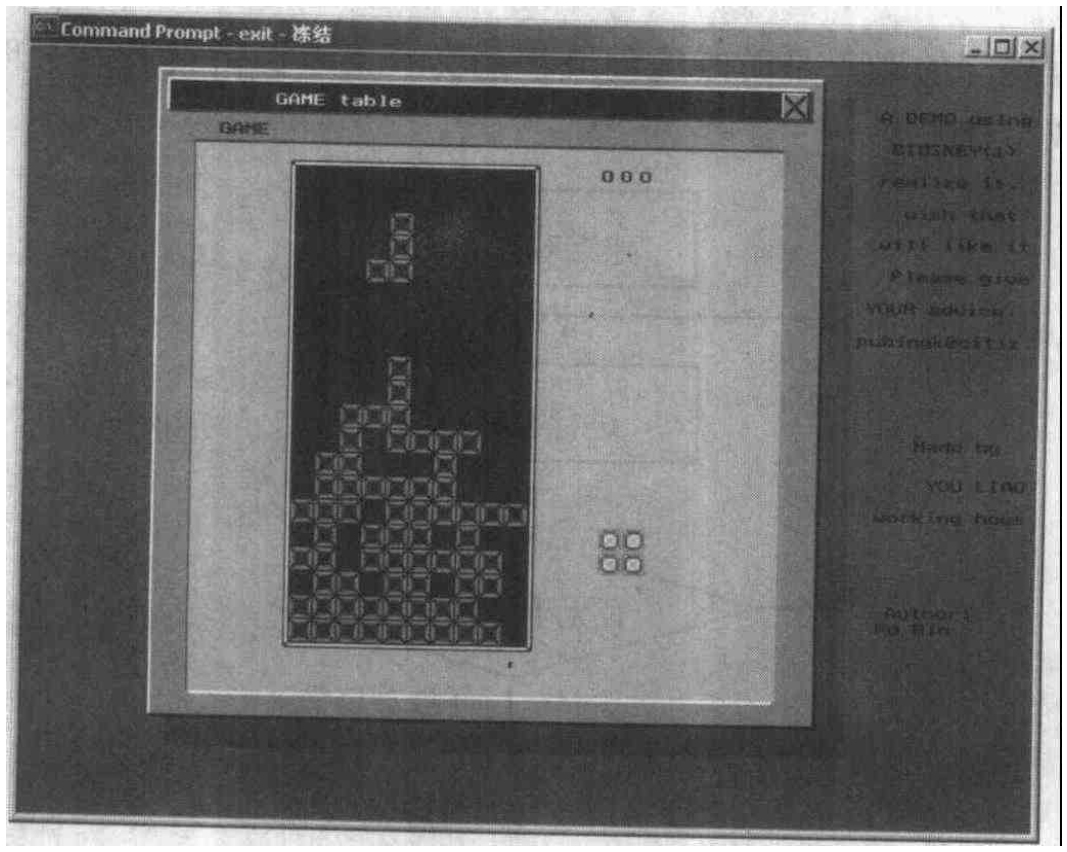


图 F

（1）游戏开始，初始设置（这个过程设置了所有的变量初始值）；

（2）进行一个方块下落或者消去的动画（这个过程只负责重画屏幕）；

（3）然后察看是否有用户发出的旋转、下落、左右移动或退出游戏的指令（这个过程接收信息，同时存入输入变量）；

（4）此后如果有用户指令根据其指令计算出块的下一个形状、位置，同时也按照游戏本身设置的下落速度进行叠加计算并且判断是否到达底部，如果到达底部再判断是否可以消去和是否死亡（这个过程改变各种变量的值）；

（5）计算完毕后，根据结果进行下一次下落或消去的动画（回到步骤 2），如此循环往复直到游戏结束（这个过程就是一个 while 语句之类的循环）。

对应上面 5 个步骤，再以一个简单的 C 语言伪程序为例：

```
#include...
```

```
#define...
void main(void)
{
int a,b,c;                  //步骤 1：相当于设置初始值
 a=1;
 b=2;
 c=3;
 while(a!='q')
 {
   printf("%d",c);  //步骤 2：相当于重画屏幕
   a=getch();       //步骤 3：相当于响应输入设备
   c=a*b;           //步骤 4：相当于重新运算变量值
 }                  //步骤 5：相当于如果 a 不等于 q 就继续循环步骤 2-4
}
```

面向对象编程思路的融入是游戏编程流程的又一重点。再来看一下程序设计，原先的流程可以被理解成结构图 G。

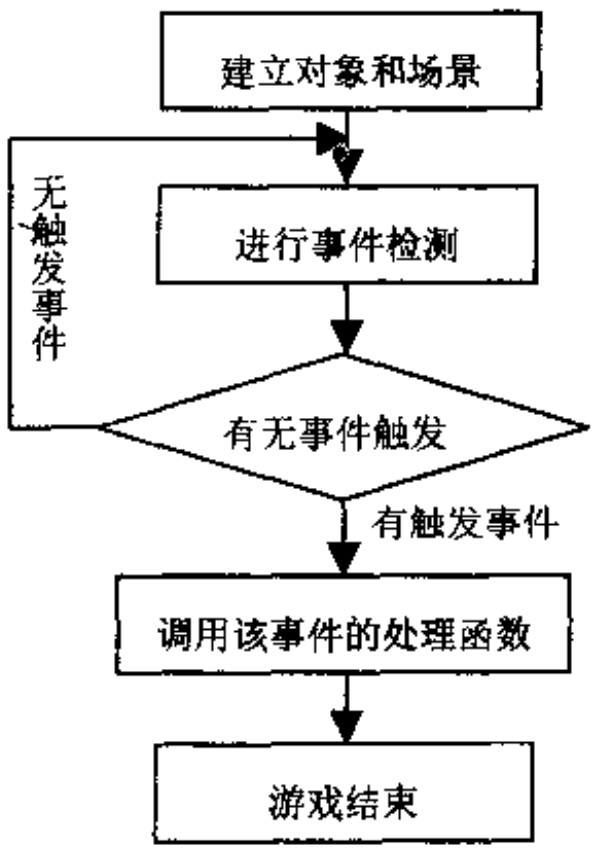


图 G　结构图

这和前面游戏流程并不矛盾，只是换个角度看问题。表 A 是对象思路中的每个环节和游戏流程的四大部分进行对应关系。

表 A　流程与对象

游戏流程	对象思路
动画	对象事件函数中的对象动画处理
响应	事件检测（也有的加上对象函数中的响应部分）
运算	对象事件函数中的运算
循环	事件检测循环

一个是流程明晰，一个是流行的对象化思路，那么到底我们按照哪种思路来编写程序呢？

由于我们使用的是 TC 平台，所以本书的思路是，在流程化语言的基础上加入面向对象思路。这样的好处是能让流程化语言也拥有一定的面向对象思路容易扩充、维护的优点。

本节就是给了你一套写游戏最简单的模版。无论你要写怎样的游戏，都将它分成动画、响应、运算和循环四部分,外加一个初始设置。只要记住给你的那个 C 语言伪程序，套用之后再一步一步改写，一个游戏就出来了。怎么样，解牛的顺序知道了吧！

3. 游戏的设计步骤

软件工程课告诉我们任何一个软件的实现都要经过以下几个步骤：

构思：你的创意，具体的内容规划；

设计软件结构：用简洁高效的数据结构实现你的创意；

编写程序：将你的数据结构用程序语言来表达出来；

调试、测试：对程序的系统可用性、模块功能进行测试，检验程序中错误和边缘问题，同时修改程序不符人意的部分；

完善、改进和推出：使游戏更加好玩、更便于操作；

游戏设计事实上也是软件的设计，然而它又有自身的一些特点。比如，游戏强调有极其新颖的创意，要对游戏的每个细节都有准确、详尽的描述和表现等。游戏编程的步骤可以归纳为如下几点。

（1）创意阶段

良好的创意。对创意进行完善，设计出游戏的完整过程并画出流程图，并将游戏分解为若干层次。

对每个层次进行详尽的描述，包括其中的人和物以及每个层次的实现目标。

对每个层次设定规则，包括人物的移动、操作者的权利、游戏的奖励和惩罚以及周边的一切环境、音效和帮助。

（2）规划阶段

使用的图形模式。比如 BIOS 中断 10H 的 13H 模式，即 200*320 像素 256 色。

确定图形、动画的复杂程度。比如用二维实现还是三维。

考虑使用游戏函数库中的哪些函数来实现游戏创意中的那些要求。

（3）周边准备

制作人物、环境的图像文件。用绘画软件制作*.JPG 或*.BMP 文件。

设计人物、环境的声音文件。用录音机制作*.WAV 文件。

建立地点、发展层次，人、物品、对话和奖励惩罚的数据库文件。例如用 Foxbase、dBase 或者 Foxpro 数据库软件建立*.DBF 文件。

（4）细部实现

图像实现函数、图像动画函数、声音播放函数、输入装置驱动和功能函数以及存盘、读盘函数。

（5）模块实现

数据结构系统、图像、动画系统、输入/输出系统、人工智能系统、游戏循环系统、用

户界面系统以及声音系统。

（6）整体完善

美工、音效的提高；游戏动画、运算效率的提高；游戏功能和完整性的扩展；游戏的文件大小的缩减。

这只是一个大的方向性原则，具体的内部设计步骤根据不同游戏的情况和个人风格进行增减或调整。然而关键的是要编写一个好的游戏一定要按照软件工程的步骤，从构思到结构到编程一步一个脚印；切不可贪快——一上来就写，到头来程序写到一半写不下去，前功尽弃。好的程序员才可能成为好的游戏编程人员。庖丁要顺骨解牛，不可乱来！

目　录

第 1 章　猜数字游戏 1
1.1　游戏创意 1
1.2　游戏规划 2
1.3　程序实现 4
1.4　游戏调试 6
1.5　文本模式游戏制作 8
1.5.1　文本窗口函数 9
1.5.2　INT10 中断功能 11
1.6　本章小结 11
第 2 章　用 C 语言函数库画图 12
2.1　设置图形模式 12
2.2　在图形模式下绘图 14
2.2.1　点 14
2.2.2　线 14
2.2.3　填充 15
2.3　在图形模式下写字 15
2.3.1　文本属性设置 15
2.4　独立图形程序的建立 16
2.5　赛车的完整图画 17
2.6　本章小结 18
第 3 章　简单动画 20
3.1　实现动画思路 20
3.2　屏幕保存与恢复 22
3.3　重画动画实例 24
3.4　简单动画实现 26
3.5　用异或实现赛车动画 29
3.6　本章小结 32
第 4 章　简单图形游戏 33
4.1　从动画到游戏 33
4.2　简单用户响应 34
4.3　接收用户信息 36
4.4　配上其它东西 39
4.4.1　配上声音 39
4.4.2　加入片头和片尾 39
4.4.3　使用随机数 41
4.5　赛车游戏 42
4.6　本章小结 43
第 5 章　图形模式 44
5.1　显示适配器与显示模式 44
5.1.1　显示适配器 44
5.1.2　显示模式 45
5.2　图形模式 13H 45
5.3　调用 BIOS 中断 10H 47
5.4　用汇编设置模式 47
5.4.1　使用汇编文件 47
5.4.2　行内汇编 48
5.5　本章小结 49
第 6 章　二维图形 51
6.1　基本图形 51
6.1.1　直接写屏 51
6.1.2　直接画点 52
6.1.3　直接画线 53
6.1.4　直接画多边形 56
6.2　图形函数优化 57
6.3　更多图形 59
6.4　本章小结 63
第 7 章　中文显示 64
7.1　文字显示原理 64
7.2　西文显示 65
7.2.1　使用 ROM 字符集 65
7.2.2　使用西文字库 67
7.3　中文平台下文字显示 69
7.3.1　汉字显示方法 69
7.3.2　中文平台判别 69
7.4　西文平台下中文调用 70
7.4.1　hzk16 中文字库文件 70
7.4.2　hzk24 中西文共显 73
7.5　小字库、无字库技术 73
7.5.1　小字库技术 74
7.5.2　无字库技术 80
7.6　中文特效 83
7.6.1　多字体显示 83
7.6.2　文字格式显示 83
7.7　本章小结 84

第 8 章　图形文件 85
8.1　bmp 文件调用 85
8.1.1　bmp 文件结构 86
8.1.2　256 色 bmp 文件显示 88
8.2　pcx 文件调用 93
8.2.1　pcx 文件结构和编码 93
8.2.2　pcx 文件显示 96
8.2.3　播放 pcx 文件 98
8.3　ico 文件显示 99
8.3.1　ico 文件结构 99
8.3.2　ico 文件显示 102
8.4　本章小结 105
第 9 章　动画原理 106
9.1　动画技术分类 106
9.2　重画技术 107
9.2.1　直接重画 107
9.2.2　缓冲技术 108
9.3　异或技术 110
9.4　调色板技术 112
9.4.1　调色板寄存器 112
9.4.2　调色板动画原理 115
9.4.3　调色板动画举例 116
9.5　拉屏技术 120
9.6　适用环境和效率 125
9.7　本章小结 127
第 10 章　子画面技术 129
10.1　子画面概述 129
10.1.1　子画面 129
10.1.2　子画面结构 131
10.1.3　面向对象 133
10.2　显示子画面 135
10.3　子画面运动 140
10.4　背景问题 141
10.5　子画面游戏 144
10.6　子画面绘制 151
10.7　本章小结 152
第 11 章　文件操作 154
11.1　文件基本操作 154
11.1.1　建立、打开和关闭 154
11.1.2　读取和写入 155
11.2　游戏进度文件 155
11.2.1　两种方法 156
11.2.2　保存进度文件 157
11.2.3　读取进度文件 159
11.3　游戏数据文件 161
11.4　dbf 文件 163
11.4.1　dbf 文件结构 163
11.4.2　dbf 文件读取 164
11.5　本章小结 165
第 12 章　声音技术 166
12.1　PC 喇叭发声 166
12.1.1　发声系统 166
12.1.2　PC 喇叭播放歌曲 167
12.1.3　扬声器背景音乐 168
12.2　声卡技术 169
12.2.1　DSP 简介 169
12.2.2　DSP 端口寻找 170
12.2.3　写 DSP 170
12.3　播放 wav 文件 171
12.3.1　WAV 文件格式 171
12.3.2　WAV 文件播放 172
12.4　WAV 背景音乐 173
12.5　本章小结 173
第 13 章　内存技术 175
13.1　常规内存 175
13.2　内存结构 176
13.3　XMS 技术 177
13.3.1　XMS 基本知识 177
13.3.2　XMS 基本函数 177
13.3.3　XMS 调用基本程序 177
13.3.4　将中文字库调入 XMS 178
13.4　EMS 技术 180
13.4.1　EMS 基本知识 180
13.4.2　EMS 调用基本程序 181
13.4.3　将中文字库调入 EMS 183
13.4.4　全方位拉屏 184
13.5　本章小结 196
第 14 章　接口技术 197
14.1　键盘 197
14.1.1　键盘读取 197

14.1.2 同时按下问题.....200
14.1.3 模拟按键.....201
14.1.4 清空键盘缓冲.....201
14.2 鼠标.....202
14.2.1 鼠标基本函数.....202
14.2.2 改变鼠标形状.....204
14.2.3 用 pcx 图像做鼠标.....209
14.3 串口.....211
14.3.1 串口基础.....211
14.3.2 利用串口传输文件.....216
14.3.3 两机坦克对打例程.....219
14.4 本章小结.....226
第 15 章 界面技术.....228
15.1 界面对象的结构.....228
15.1.1 对象的结构分析.....228
15.1.2 对象的初始化.....230
15.1.3 界面设计与分析.....231
15.2 对象绘制函数.....232
15.2.1 填充矩形绘制函数.....232
15.2.2 立体按钮绘制.....233
15.2.3 窗体、按钮和菜单绘制.....233
15.3 使用链表.....238
15.4 对象事件函数.....239
15.4.1 按钮的基本动作.....239
15.4.2 菜单的基本动作.....240
15.5 进行事件检测.....243
15.6 界面例程.....244
15.7 游戏实例.....245
15.7.1 DOS 游戏界面设计.....245
15.7.2 将界面插入游戏.....247
15.7.3 构建个性化界面.....250
15.8 本章小结.....251
第 16 章 其他问题.....252
16.1 TSR 驻留.....252
16.1.1 TSR 基本知识.....252
16.1.2 时钟驻留.....253
16.1.3 热键驻留.....253
16.2 简单病毒.....253
16.3 OOP 应用.....254
16.4 各类游戏编程思路.....256
16.4.1 桌面游戏编程思路.....256
16.4.2 视频对战游戏编程思路.....257
16.4.3 魂斗罗类游戏编程思路.....259
16.4.4 玛丽、赛车类游戏编程思路.....260
16.4.5 RPG 游戏编程思路.....262
16.5 本章小结.....263
第 17 章 游戏例程.....265
17.1 建立通用游戏函数库.....265
17.2 游戏创意.....265
17.3 游戏规划.....268
17.3.1 详细设计.....268
17.3.2 程序流程设计.....271
17.4 程序编写.....272
17.4.1 文件清单.....272
17.4.2 进度文件.....272
17.4.3 图片文件.....272
17.4.4 数据文件.....273
17.4.5 代码文件.....276
17.5 游戏场景.....276
17.6 本章小结.....281
附录 A 游戏函数库.....282
附录 B 简单数据库.....297
B.1 数据库要求.....297
B.2 详细设计.....298
B.3 模块设计.....300
B.3.1 输入.....300
B.3.2 检查.....303
B.3.3 显示.....307
B.3.4 删除.....309
B.3.5 插入.....317
B.3.6 查找.....322
B.3.7 修改.....329
B.3.8 排序和交换节点.....337
B.3.9 保存.....343
B.3.10 读取.....344
B.3.11 清空.....346
B.4 程序代码.....346
B.5 通用数据库设计.....397

第 1 章　猜数字游戏

本章导读

猜数字的游戏规则是，电脑在 0~9 这 10 个数字中，任意不重复地选择四个排列成四位数，然后让玩的人猜使用的是哪四个数字和数字在第几位；A 代表数字对了但位置不对；B 表示数字和位置都对了，A、B 前的数字表示处于两种情况下的数字个数；举例，电脑给你猜的数字是 2943，你猜 2893，电脑就显示 2A1B。

本章重点

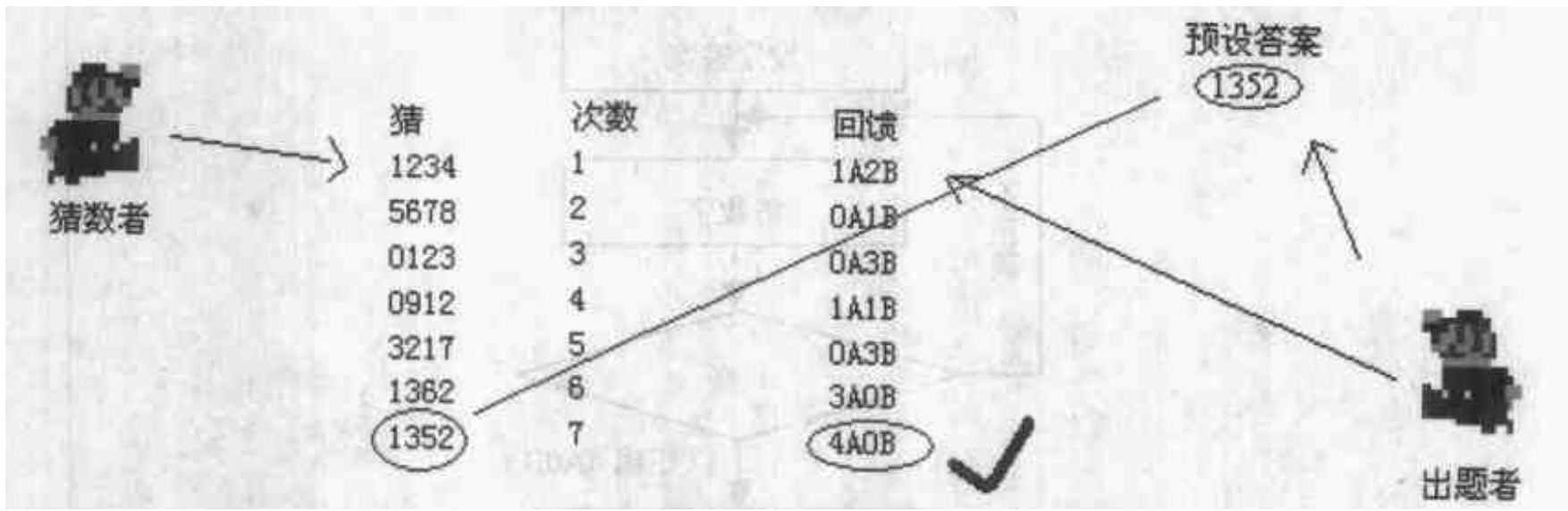
通过实例了解游戏设计的完整过程：

（1）游戏创意、游戏规划、程序实现、游戏调试；

（2）使用文本窗口函数和 INT10 中断（视频服务程序）来帮助文本游戏制作。

1.1　游 戏 创 意

首先我们来模仿以下游戏的进行过程，如图 1-1 所示。



如果 1 个数字对了但位置不对就用 1B 表示，如果 1 个数字和位置都对了就用 1A 表示

图 1-1　猜数字游戏示意图

前提：

（1）两个游戏者：一个出题（以下称为出题者），一个解题（以下称为猜数者）；

（2）两张白纸：一张写答案数字（给出题者写），一张写猜测的数字（给猜数者写）；

出题者预设答案：1352

猜数过程见表 1-1。

表 1-1　猜数字游戏过程

次数	猜（猜数者）	回馈（出题者）
1	1234	1A2B
2	5678	0A1B

（续表）

次数	猜（猜数者）	回馈（出题者）
3	0123	0A3B
4	0912	1A1B
5	3217	0A3B
6	1362	3A0B
7	1352	4A0B（正确）

事实上猜数者是根据每次出题者给出的反馈来猜测下一个数字，从而越来越靠近答案。而猜数者使用的次数越少则猜数循环次数越少，积分越高。

猜数字：出题者在 0~9 中任选不重复四个数排列成四位数，然后让猜数者每次通过从 0~9 中任选四个数字来猜使用的是哪四个数字和这四个数字各在哪个位置？如果 1 个数字对了但位置不对就用 1B 表示，如果 1 个数字和位置都对了就用 1A 表示；比如，给你猜的数字是 2943，你猜 2893，回答是 2A1B，因为 2 和 3 的数子和位置都对了，而 9 的数字对了位置没有对。这样循环往复，在每次猜数和猜的正确性信息基础上，得出最终答案。看谁猜的次数最少谁就积分越高。

通常，出题者会在一张小纸片上写好他想给别人猜的数字。而猜数者在另一张纸上写上每次的答案并给出题者看，出题者把每次猜的情况回馈在答案后面。如此循环往复直到得出正确答案。游戏进行流程如图 1-2 所示。

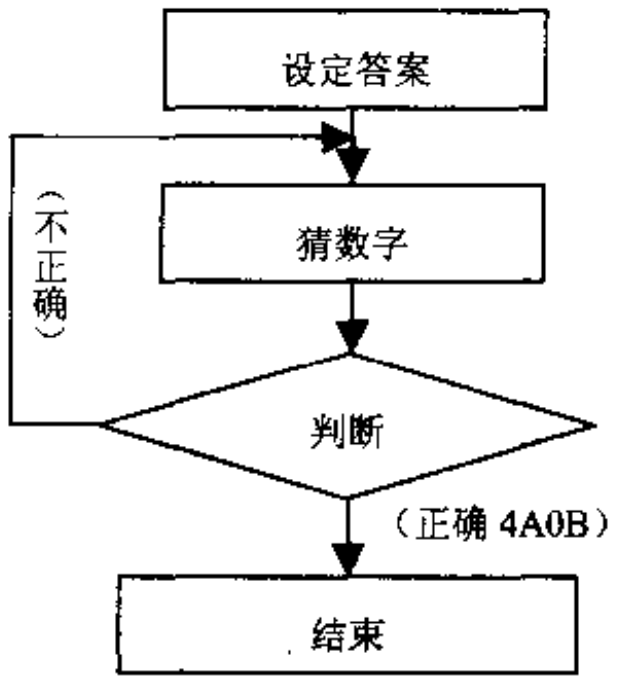


图 1-2　猜数字执行流程

由此可见，创意阶段说到底就是搞清楚到底要做一个什么游戏和这个游戏最基本的进展流程。比如这个游戏用口授教别人如何玩并不很难，可使用文字表达真的不容易，倘若我们连自己要做什么游戏都不是非常清楚，那么到真写程序的时候一定会出很多问题。事实上对游戏运行的流程描述，一方面进一步帮助我们了解游戏规则，另一方面也为今后的程序设计流程作了铺垫。

1.2　游 戏 规 划

由于当时我写这个游戏时候的情况和大家差不多，都是一个编程初学者。于是，我只

可能选择用文本模式；并且由于当时的能力所限，又决定不使用任何例如图形、菜单之类的界面支持；换句话说，当时我的规划很简单，就是将这个游戏用最简朴但仍然可懂的方法表达出来。

此外，为了尽量简化游戏编写难度我还对游戏作了以下调整：

（1）由程序固定被猜数字；

（2）由玩者给出猜测答案，程序给出反馈；

（3）不包括由玩者出题；

（4）暂不进行非数字和非四位数字的判别等周边工作；

（5）不提供任意终止游戏的功能，必须将游戏进行到底；

（6）不使用子函数（当时也不会写）；

这样一来，所要做的那部分程序更加清楚了，对游戏的编写进行一个简单的、不全面的分析之后，我当时列出了以下几个难点：

（1）几 A 几 B 的判断问题之算法；

（2）玩者所猜数字从键盘输入的函数调用；

（3）游戏循环出口的设定；

（4）被猜数字随机数产生的函数调用；（可选）

针对以上 3 个难点，我在思路中给出了一点解决方案：

（1）对于几 A 几 B 问题的算法实现，我以为不会非常简单，必须到程序中不断调试才可能完善。基本思路是，将给出的被猜数字变量的每个位和每次玩者输入的猜测答案变量的每个位进行一一比较，结合给双方各设定一个当前位标志给出每次 A、B 的数量。具体如下图 1-3 所示。

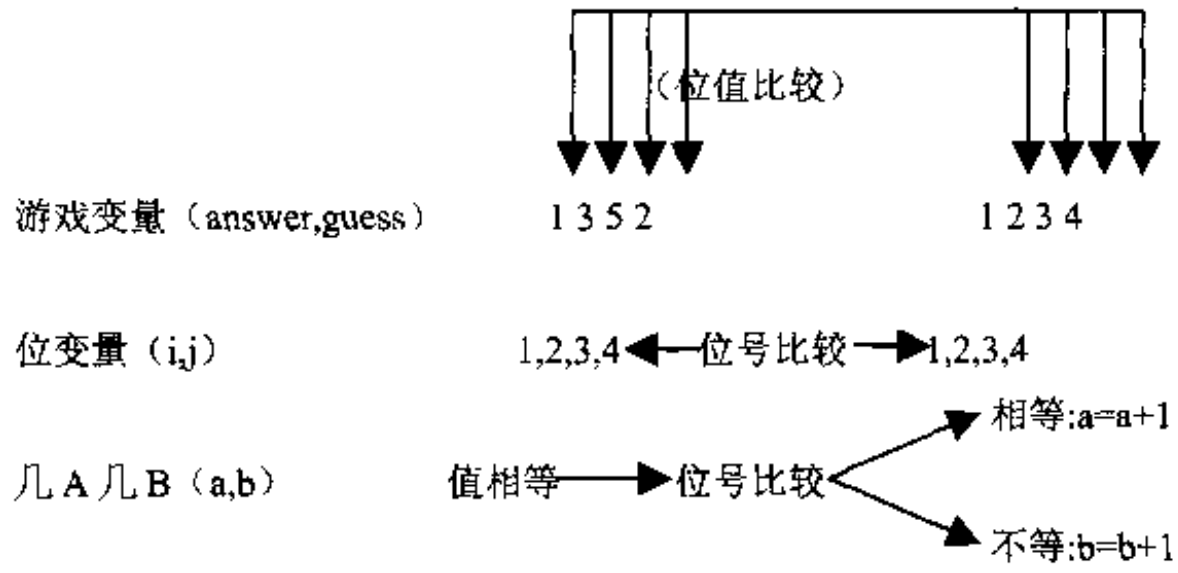


图 1-3　数字循环与判断

（2）对于输入函数的选用，一个是 getchar 函数（字符输入函数），一个是 scanf 函数（格式输入函数）。这里我们要的是一个数字，当然使用允许返回数字类型的 scanf 函数。

（3）游戏循环出口的问题，其实可以看作 while 语句中设定什么表达式的问题。我之前已经忽略了“任意终止游戏的工作”，于是这里只要判断玩者是否猜对了答案就可以了。问题一下子便简单了，因为猜对答案就是意味着 4A0B 的情况，也就是说那个变量 a=4 就

是我们要的表达式。

在规划阶段，我们不仅将游戏创意明确是为了程序编写要求和编写计划，同时对在程序编写前遇到的难点进行分析和提出大致解决方案，这些方案对程序编写是极为重要的，因为在程序设计阶段将变成具体的算法、语句和表达式。

这里值得一提的是，在规划阶段我们应该对关键、重要变量十分了解。有几个关键变量，分别代表什么，在程序中起什么作用。

1.3　程 序 实 现

大家可能急不可耐的想看程序了吧。在这之前仍然还有几件事情需要做：

（1）决定使用哪些变量

answer[4]：用数组保存四位被猜数字的每一位的值；

guess：接收玩者每次输入的数字和一点其他用处（稍候程序中可见）；

a,b：保存“几 A 几 B”，也就是每次玩者给出猜测答案的反馈；

times：猜了几次；

i,j：分别用来标示当前正在比较的被猜数字答案的位号和猜测答案的位号。

很巧，我发现在这个程序中使用到的变量都是 int 型的。当然，我不能保证是否遗漏了变量，很可能程序中会使用到更多的变量。

（2）程序流程设计

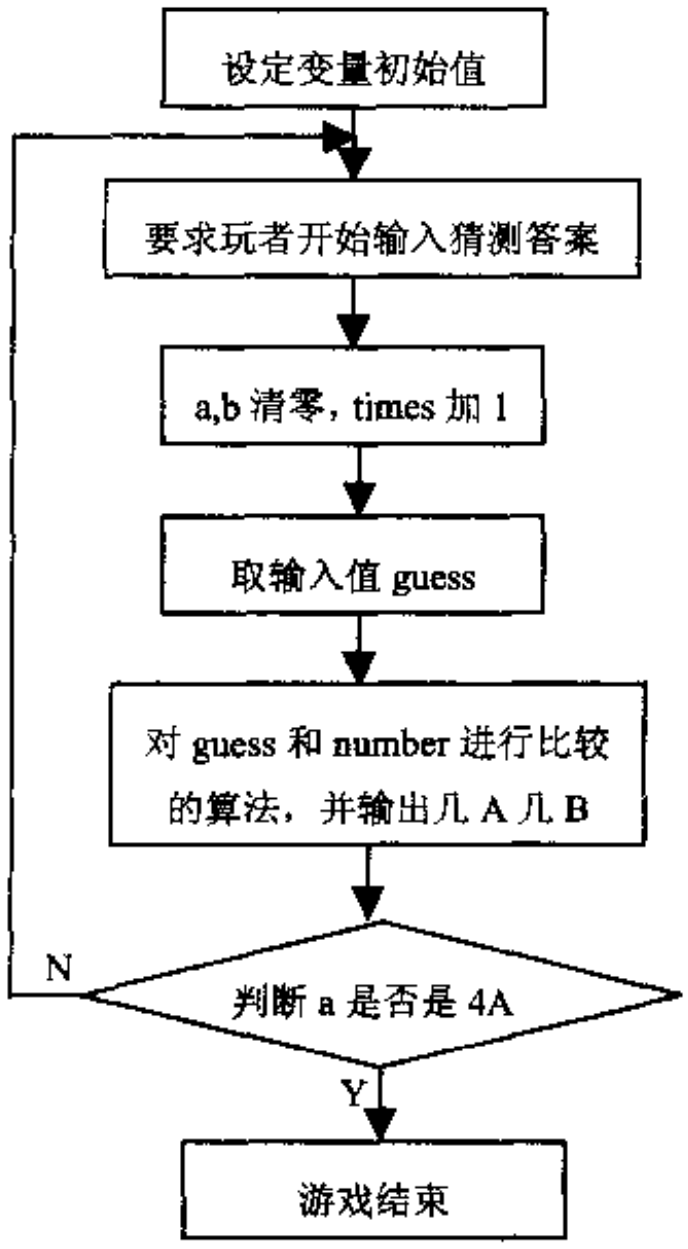


图 1-4　猜数字流程

（3）通用游戏编程模板

游戏编程模版在前面已经介绍过，在这里重新提出来的原因，是希望大家在写游戏的时候始终记着这个模板。万变不离其踪，绝大多数的游戏都逃不过这个格式。

```
#include...
#define...
void main(void) {
    int a,b,c,                    //设置初始值
    a=1;
    b=2;
    c=3;
    while(a!='q') {//循环直到退出游戏标志出现
        printf("%d",c);      //相当于重画屏幕
        a=getch();           //响应输入设备
        c=a*b;               //相当于重新运算变量值算法
    } }
```

以下给出“猜数字”游戏的完整程序 guess1.c：

```
#include <math.h>
#include <stdio.h>
void main(void) {
  int answer[4]={1,3,5,2},guess,a,b,times=0,i,j;//初始化被猜答案为 1352
    printf("please guess\n");
    while(a!=4) { //进入猜数字循环
      a=0;
      b=0; //每次猜测前设定 A、B 都为 0
      times++;//猜测次数加 1
      printf("%d      ",times);//显示猜测次数
      scanf("%d",&guess);//从键盘读取本次猜测的四位数字
  for(i=3;i>-1;i--) { //进入猜测和正确答案比对循环
    for(j=0;j<4;j++) {
      if((int)(guess/pow10(i))==answer[j]) {
      //将每一位猜测数字比对，pow10(i):取 10 的 i 次方的函数
        if(i+j==3)//如果位置相同 A 加 1
          a=a+1;
        else//如果位置不同 B 加 1
          b=b+1; }     }
      guess=guess-(int)(guess/pow10(i))*pow10(i);//比对下一位猜测数字
    } printf("      %dA%dB\n",a,b);//显示本次猜测的 A、B 正确性
} }
```

游戏运行过程的结果：

```
please guess
1    1234（玩者输入）
     1A2B
2    5678
     0A1B
3    0123
     0A3B
4    0912
     1A1B
5    3217
     0A3B
6    1362
     3A0B
7    1352
     4A0B
```

由于有了创意阶段、规划阶段的良好铺垫和那个结构近乎一样的模板，程序写的相当顺利。只是在“几 A 几 B”判断的算法问题上面稍微动了一下脑筋。这里对这个算法部分再进行详细的分析：

```
for(i=3;i>-1;i--) {//被猜数从 10 的 3 次方开始整除，以取得最高位值，一直除到 0 次方
   for(j=0;j<4;j++) {//真实答案从最高位（数组最前面）开始取，直到数组最后
      if((int)(guess/pow10(i))==answer[j]) {//比较两者相应位是否相同
         if(i-j==3)//比较两者所取的当前位是否同位
            a=a-1;//如果同位则 A 加一
         else b=b-1;//如果不同位则 B 加一
      } }
   guess=guess-(int)(guess/pow10(i))*pow10(i);//被猜数舍去当前最高位
}
```

程序设计阶段，其实并不是一上来就写程序，还需要先设计好你所能预计的所有变量，并且至少在心中有一个程序流程。对于游戏编程，这个流程通常可以套入刚才那个通用游戏程序模板。接下来就可以很快的写出程序框架了。

1.4　游 戏 调 试

不知道大家是否运行了刚才那个游戏，的确当我们正确输入四个数字的时候它能够给我们满意的回答。可是我还是发现很多规划阶段预计和程序设计阶段未预计到的问题：

（1）输入字母将死循环，没有设置输入报错；

（2）游戏未规划任意终止功能，给玩者带来了极大的麻烦；

（3）未禁止输入大于 4 位的数字和负数；

（4）未提供随机产生被猜数字功能，使游戏只能玩一次。

针对以上问题，对程序改写使得：

（1）输入字母不会导致死循环，而是表示退出游戏；

（2）禁止输入大于四位的数字和负数；

（3）每次重新运行程序都提供新的被猜数字。

改写代码 guess2.c 如下：

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
void main(void) {
    int answer[4],guess=0,a,b,times=0,i,j;
    for(i=0;i<=3;i++) {
        randomize();//随机数发生器
        answer[i]=rand()%10;//产生模为 10 的随机数
        for(j=0;j<i,j++) {
            while(answer[i]==answer[j]) { //判断是否和之前产生的随机数相同
                answer[i]=rand()%10;//如果相同，则重新产生
                j=-1;
            } } }
    printf("please guess\n");
    while(a!=4) { a=0;
        b=0;
        times++;
        do {     printf("%d     ",times);
            scanf("%d",&guess);
        if(guess==0)//输入的是字符
            exit(0);
    }while(guess<0||guess>9999);//要求输入规定范围内的数字
    for(i=3;i>-1;i--) {
        for(j=0;j<4;j++) {
            if((int)(guess/pow10(i))==answer[j]) {
                if(i+j==3) a=a+1;
                else b=b+1;
            } }
        guess=guess-(int)(guess/pow10(i))*pow10(i);
```

```
        } printf("      %dA%dB\n",a,b);
    } }
```

对于游戏编写者，程序的实现应该是最简单的一件事情。因为所有的人只要它学过一点程序编写，便总能够在反复的失败中实现程序（即便用的方法很傻）。最难的是在程序开始前设计一个好的算法和游戏数据结构以及在程序初步实现后进行调试，将所有可能的问题都避免掉。

无论是从我设计的这个游戏解决方案还是从我写的这个改进的程序看，其实还是有很多问题和可以优化的地方。希望大家能够根据这个游戏命题，按照游戏设计的步骤自己来试试。

这一章最重要的是想告诉大家在游戏编程中，程序实现并不是最重要的部分。大家千万不要一上手就写，而是应该按部就班地按照游戏实际步骤做。磨刀不误砍柴功——设计好的算法、数据结构和解决方案往往可以大大提高游戏编写的速度。

1.5　文本模式游戏制作

文本模式下，共有 80 列*25 行个文本单元，每个单元由显存中 2 个字节对应，前一个字节放 ASCII 码字符和后一个字节用来规定该字符的颜色、闪烁等属性。客观上就决定了文本模式所能传达的信息量远远小于图形模式。

文本模式游戏事实上是最简单的游戏。从游戏编写者角度讲，简化了图形、动画等一系列极为重要的游戏编程部分，考虑的问题主要就是算法，这使得游戏制作变得相当简便，极其适合游戏编程初学者；而对于玩游戏的人来说，这类游戏最没有意思，但也有一些强调思维分析、推理能力的经典文本游戏依然可以吸引玩者。

从文本模式游戏的选题来说，通常一定不能找动作游戏，至少我没有看到过文本模式的“街霸”，不过我倒是出于算法实现角度做过一个文本模式的“俄罗斯方块”；最好找那些近似依靠数学思维能力、需要动脑筋的游戏或者是只有极为简单横竖线条的类图形游戏，因为形式的局限，所以还必须找这类游戏中最风靡的才会有人愿意玩。“猜数字”无疑是其中最有代表性的一个。

文本模式游戏的流程结构较其它形式游戏更显清晰，说白了就是一个带有输入、运算和输出的循环。这里我要第三次提出那个通用游戏模板（具体请见 1.4 节），请大家务必重视它在游戏编程中不可替代的重要地位。事实上对于文本游戏，这个模板基本不需要任何变动即可按游戏流程图的每个层次要求将模板中每个语句直接替换为具体的语句群。

文本模式游戏制作要求编程者对数学函数 math.h、标准输入输出函数 stdio.h 相当熟悉，此外还需要有良好的数据结构和算法设计功底。如在“猜数字”游戏中对“几 A 几 B”的问题解决，可以理解成：将答案和猜测两个变量的每个位比较相等，若相等对各自位比较相等，从而得到几 A 几 B 的算法。这种算法我们在从前的 C 语言课程习作中经常碰到，无非就是使用两个 for 循环再加上两个 if 判断以及对输入数字进行必要的运算变形。如果不懂得算法，可能就会用很多语句去表达，而且往往还会有许多错误，到最后连做下去的信心都没有了。所以这里建议大家有空多看看数据结构和算法的书，这可以大大提高你写游

戏的速度和质量。

如果想在文本模式下面做更好的游戏，还要涉及到文本模式下直接写屏操作、BIOS 的 INT10 关于屏幕光标操作的中断和界面菜单制作。

1.5.1　文本窗口函数

通常，文本模式下背景是黑色的；而字符是单一的白色；此外，所有字符通常都是从屏幕的最左下角开始显示。为了使文本游戏显得相对生动，我们希望能够丰富字符和背景的色彩以及使输出的字符位置更加灵活。

conio.h 头文件中声明了许多针对文本模式屏幕处理的函数，主要包括文本窗口建立、字符、背景颜色的设置、窗口文本的清除和输入输出等函数；有关函数请查阅所附光盘的“book\附录 C”。

以下是一个简单的窗口范例 window.c：

```
#include <conio.h>
int main(void)
   char ch;
   clrscr();
   window(10,10,70,11);
   textattr(128+YELLOW+(GREEN<<4));
   cprintf("Please select a level (1,2,3) you want to play: ");
   ch = getche();
   return 0; }
```

我们在设置文本属性的时候使用了以下语句：

```
textattr(128+YELLOW+(GREEN<<4));
```

本章开始的时候说过，文本模式（80*25 文本单元）下每个文本单元由两个字符组成。第一个字符存放字符的 ASCII 码，而后一个字符内存放了 3 个相关属性。那么到底是那 3 个属性呢？

文本属性包括了文字颜色、背景颜色和是否闪烁 3 个属性。从刚才设置文本属性的语句行我们可以看到 128 表示闪烁、YELLOW 表示文本颜色、GREEN<<4 表示背景颜色。在 8 个位的一个字符中它们具体的存放规则如下：

位	8	7	6	5	4	3	2	1
含义	是否闪烁	背景颜色			文本颜色			
具体值	1 闪 0 不闪	0~7 颜色			0~15 颜色			

这就是为什么 GREEN 要左移 4 位的原因，因为它表示背景颜色；而 YELLOW 只要取 0~15 之间的数就可以表示文本颜色了；128 正好是第 8 位为 1，也就是表示闪烁。

以下对“猜数字”游戏进行一些修改，加入了一些文本窗口、色彩和背景，使游戏效果有所提高 guess3.c：

```
#include <conio.h>
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
void main(void) {
    int answer[4], guess=0, a, b, times=0, i, j;
    clrscr();
    for(i=0;i<=3;i++) {
        randomize();
        answer[i]=rand()%10;
        for(j=0;j<i;j++) {
            while(answer[i]==answer[j]) {
                answer[i]=rand()%10;
                j=-1;
            } } }
window(0,0,40,11);//设定文本窗口，从(0,0)到（40,11）
textattr(YELLOW+(GREEN<<4));//设定文本属性，绿底黄字
cprintf("please guess\n");
while(a!=4) { a=0;
    b=0;
    times++;
    do { window(10,0,40,21);
        cprintf("\r%d     ",times);
        cscanf("%d",&guess);
        if(guess==0)
            exit(0);
    }while(guess<0||guess>9999);
    for(i=3;i>-1;i--) {
        for(j=0;j<4;j++) {
            if((int)(guess/pow10(i))==answer[j]) {
                if(i+j==3) a=a+1;
                else b=b+1;
            } }
        guess=guess-(int)(guess/pow10(i))*pow10(i);
    } window(20,0,40,31);
    cprintf("\n\r       %dA%dB\n",a,b);
} }
```

图 1-5 是猜数字游戏屏幕图像。

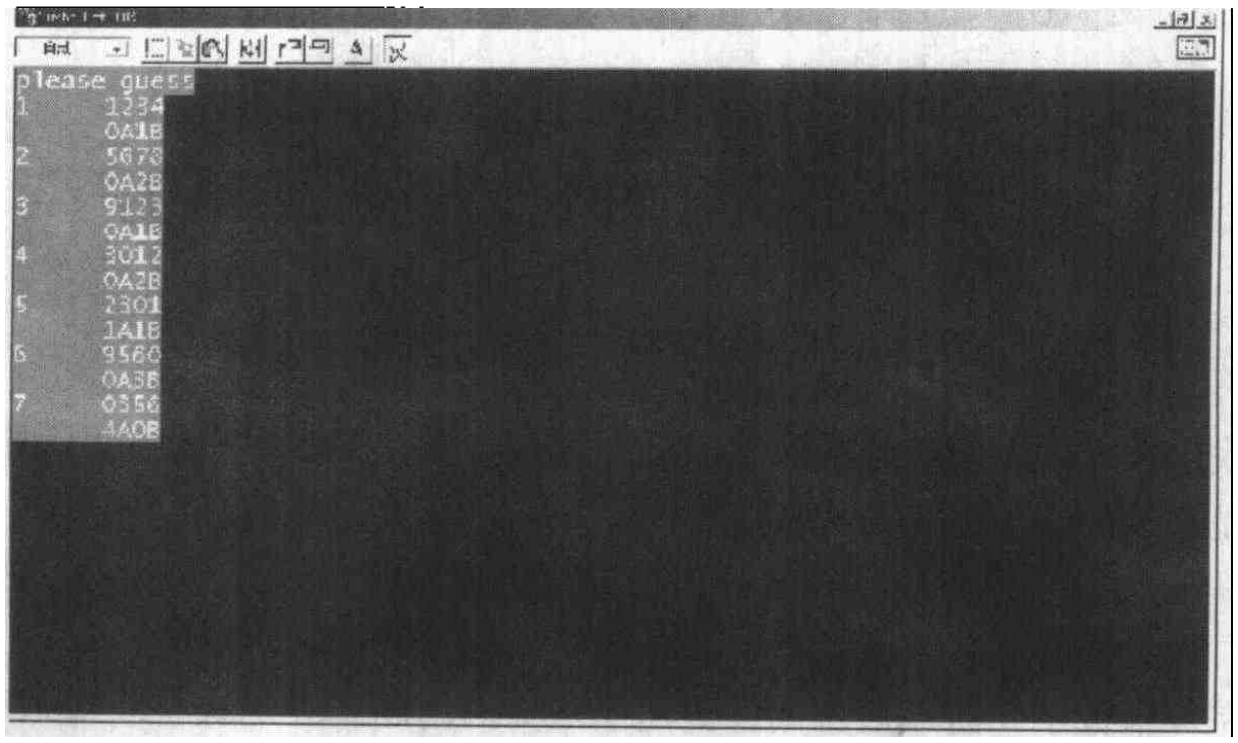


图 1-5　猜数字游戏示意

1.5.2　INT10 中断功能

在文本模式下写游戏，除了可以使用上一节介绍的 TC 窗口、屏幕库函数，我们也可以直接调用 BIOS 中断 INT10 提供的强大的光标、屏幕操作功能。详细的关于 INT10 中断的一些功能号的介绍请查阅所附光盘的“book\附录 D”。

一个测试 INT10 中断功能的例程 int10.c 请查阅所附光盘的“source\1”目录。

1.6　本 章 小 结

通过本章我们知道游戏制作并不是简单地写一个游戏程序，我们必须使用软件工程的设计流程来完成它。游戏设计分创意阶段、规划阶段、程序实现阶段和调试阶段。只有养成了良好的游戏设计习惯，才能保证在制作大游戏的时候游刃有余。倘若一上手就冲动的写程序会导致最终的寸步难行、中途放弃。

通过本章学习读者应该基本掌握文本游戏制作的方法：

（1）套用文本游戏模板；

（2）利用窗口函数和 int10 视频服务中断。

学后建议

制作一些文本游戏（比如文本俄罗斯方块）；

开发文本窗体、菜单、对话框界面函数；

设计简单文本编辑软件（如同 DOS 下的 edit.com）。

第 2 章　用 C 语言函数库画图

本章导读

要写游戏，必须首先学会绘制图像，而我们在 C 语言课堂中只教过用 printf()函数来输出文本，这显然不能满足绘图的要求。

许多 C 语言初学者总是问，C 语言到底是否给我们提供了图形函数？这些图形函数又是怎么用的？为什么明明找到了它们却总是说初始化图形模式不对？

当我写完“猜数字”游戏后也同样提出了这样的问题。于是我就经常到书店里面找 C 语言图形方面的书籍，最后找到一本《C 语言图形设计》。从书中我了解到，C 语言标准函数库提供了一个较为强大的图形函数库（包括：圆，正方形等图形；各类图形的填充模式和各方面属性；图形保存和显示；图形模式初始化等函数），所有图形函数都在头文件 graphics.h 中声明。

程序在包含了 graphics.h 头文件后，所有图形函数的使用都必须在图形模式下进行。而我们的默认模式都是文本模式，在这种模式下所有图形函数是无法正常工作的。所以必须先使用一个图形模式初始化函数将计算机设置为图形模式。

本章就是向大家介绍这个 C 语言图形函数库和它的使用方法。

本章重点

（1）使用标准图形函数库中 initgraph()函数让计算机进入图形模式；

（2）掌握 c 标准图形函数库中各类图形函数的使用方法。

2.1　设置图形模式

要使用 C 语言绘制图形通常首先要包含 graphics.h 头文件，它提供了大量的图形绘制函数。我们无法立即使用这些绘图函数，必须首先设置屏幕为图形模式。

要将屏幕原先默认的文本模式(80 列, 25 行字符模式)设置成图形模式，必须对显示卡进行操作。显示卡事实就是显示适配器的通称。不同的显示适配器有着不同的色彩种数和图形分辨率。即便同一显示适配器，在不同模式下工作也有着不同的色彩种数和图形分辨率。

因此，在使用图形函数作图之前，必须先使用一个 graphics.h 图形函数库提供的初始化图形模式的 initgraph()函数根据显示适配器种类将显示器设置成为某种确定的图形模式。

关于图形初始化函数的介绍请查阅本书所附光盘的“book\附录 E”。

初始化图形模式对于 C 语言新手来说的确有一些难度，很容易走弯路或者在这里卡住。当我获得《C 语言图形设计》之后，开始尝试图形编程。然而最初我无论如何都无法显示图形（即便是完全按照书中的例程写）。我仔细检查，总觉得自己没有出错。既然在程序最开始加了#include<graphics.h>，那么理所应当可以开始使用图形函数了，可是屏幕上总是出现：BGI Error: Graphics not initialized(use ‘initgraph’)。这到底是怎么一回事呢？原来，我

在初始化图形模式语句中没有将驱动程序的路径写对，导致程序无法找到图形驱动文件，从而无法在图形模式下进行工作。

以下给出一个最简化的初始化图形模式的例程可以更清楚地说明问题：

```
#include <graphics.h> //声明标准图形函数头文件
int main(void) {
     int gdriver, gmode; //定义图形驱动器变量和图形模式变量
     gdriver-DETECT; //设定图形驱动器为自动监测
     initgraph(&gdriver, &gmode, "c:\\tc"); //初始化图形模式
  line(100,100,200,200);                    //画线
     getch();
     closegraph(); //关闭图形模式
     return 0;
}
```

这里有几点值得大家注意：

（1）对于 gdriver 原本要求设置你所希望的并且计算机提供的图形驱动器，而 gmode 则是对应于这种驱动器的使用模式。但我们建议将 gdriver 设置为 DETECT，让硬件自动检测图形驱动器和模式，这将非常省力。

（2）如果我们发现 bgi 文件在 C:\tc 目录下，而当前目录在 C:\下，path 可以使用绝对路径，如："c:\\tc"；也可以使用相对路径，如："\\tc"。此外，对于相对路径来说，如果我们将此程序复制到其他目录很可能就无法正确找到图形驱动程序了（如我前面碰到的问题）。所以建议大家使用绝对路径或者将图形驱动文件复制到源程序同一个目录下面（此时就不用写路径了，因为不写路径的时候相对路径就是当前目录）。

（3）一旦初始化了图形模式我们便可以作图了，在程序中 line(100,100,200,200)语句就是在图形模式下画一条从点(100,100)到点(200,200)的直线。

（4）我们使用 closegraph()函数来退出图形状态回到默认的文本状态，其调用格式请查阅所附光盘的“book\附录 E”。

自动进行硬件测试后进行图形初始化例程 init.c 如下：

```
#include <graphics.h>
int main() {
     int gdriver, gmode;
     detectgraph(&gdriver, &gmode);//检测图形模式函数
     initgraph(&gdriver, &gmode, "c:\\tc");
  line(100,100,200,200);                    //画线
     getch();
     closegraph();
     return 0;
}
```

与前一个程序比较这里只是增加了一句 detectgraph(&gdriver, &gmode)，其效果和将 gdriver 设置为 DETECT 完全相同。然而明显第一种方法简单了很多。

2.2　在图形模式下绘图

以下给出一些 graphics.h 函数库中提供的关于图形绘制的一些函数的例子。

2.2.1　点

关于画点函数、坐标位置函数 putpixel()以及 getmaxx()等请查阅所附光盘的“book\附录 E”。

2.2.2　线

有关画线函数 line()等请查阅所附光盘的“book\附录 E”。

polypoints 通常被初始化为数组，所有下标为双数的数组都表示 x 坐标，而下标为单数的数组都表示 y 坐标，以下给出了一个简单的 5 边形例程 poly.c：

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void) {
   int gdriver = DETECT, gmode;
   int poly[12];//定义多边形顶点数组变量
   initgraph(&gdriver, &gmode, "");
   poly[0] = 0;//定义第一个顶点 x 坐标值
   poly[1] = 20;// 定义第一个顶点 y 坐标值
   poly[2] = 40;
   poly[3] = 20;
   poly[4] = 40;
   poly[5] = 40;
   poly[6] = 20;
   poly[7] = 40;
   poly[8] = 10;
   poly[9] = 35;
   poly[10] = 0;
   poly[11] = 20;
   drawpoly(6, poly);//调用绘制多边形函数，为 6 边形，顶点使用 poly 数组变量值
   getch();
   closegraph();
   return 0;
}
```

有关线型函数 setlinestyle()等请查阅所附光盘的“book\附录 E”。

2.2.3　填充

有关图形填充函数 bar()等请查阅所附光盘的“book\附录 E”。

上一节的填充方式只能对一些特定形状的封闭图形进行填充，而对于任意封闭图形的填充是无能为力的。C 语言提供了一个可对任意封闭图形填充的函数。

2.3　在图形模式下写字

在图形模式下我们仍然可以使用标准输出函数 printf()，puts()，putchar()将文本输出到屏幕。不过，C 语言图形函数中提供了一些专门用于在图形显示模式下的文本输出函数 outtext()等。有关文本输出函数请查阅所附光盘的“book\附录 E”。

2.3.1　文本属性设置

我们可以利用 setcolor()函数设置输出文字的颜色。此外，我们还可以改变文本字体的初始点、大小以及选择是水平方向输出还是垂直方向输出。

用户对文本字符大小的设置。settextstyle()函数可以设定图形方式下输出文本字体大小，但在水平和垂直方向以相同的放大倍数放大。是否可以使水平和垂直方向放大倍数不同呢？

图形函数库中提供了 setusercharsize()函数，对笔划字体可以分别设置水平和垂直方向的放大倍数。该函数的定义格式请查阅所附光盘的“book\附录 E”。

以下给出一个在图形模式下写字的综合例子 text.c：

```
#include<stdio.h>
#include<graphics.h>
int main() {
    int gdriver, gmode;
    gdriver=DETECT;
    initgraph(&gdriver, &gmode, "");
    setcolor(12);
    settextstyle(4, 0, 8);//黑体笔划字，放大 8 倍
    outtextxy(100, 100, "game over");
    setusercharsize(2, 1, 4, 1).//水平放大 2 倍，垂直放大 4 倍，此处不起作用
    setcolor(15);
    settextstyle(2, 0, 4); //小号笔划字体，放大 4 倍
    outtextxy(200, 200, "game over");
    setusercharsize(4, 1, 1, 1); //水平放大 4 倍，垂直放大 1 倍
    settextstyle(3, 0, 0);
    outtextxy(0, 300, "game over");
    getch();
```

```
    closegraph();
    return 0;
}
```

2.4 独立图形程序的建立

大家在图形模式下写好了游戏，然后 email 给其他朋友。可是对方却根本无法玩，这是怎么回事？原来，当我们在设置图形模式的时候，要求有对应的 BGI 文件（对于用 initgraph()函数直接进行的图形初始化程序，在编译和链接时并没有将相应的驱动程序*.BGI 装入到执行程序）。而我们将游戏程序 copy 给其它朋友的时候并没有将 BGI 文件 copy 给他们。于是他们根本无法进入图形模式（当程序进行到 intitgraph()语句时，从该函数中第三个形式参数 char *path 所规定的路径中去找相应的驱动程序。若没有驱动程序，则在 C:\tc 中去找，若仍没有或 TC 不存在，将会出现错误: BGI Error: Graphics not initialized (use 'initgraph'))。即便我们将所有 BGI 文件 copy 给他们，还是可能存在一个路径错误的问题。有什么好的方法么？

我们只要将 BGI 文件（图形驱动程序）也一起做到程序中来，问题就解决了。这里提供了建立一个不需要驱动程序就能独立运行的可执行图形程序的方法，以下是具体步骤(这里以 EGA、VGA 显示器为例)：

（1）在 C 语言编译器目录（就是 TC 根目录，要求 bgiobj.exe 和 bgi 文件都在这个目录）下输入命令：

```
BGIOBJ EGAVGA
```

BGIOBJ 命令将驱动程序 EGAVGA.BGI 转换成 EGAVGA.OBJ 的目标文件。

（2）在 C 语言编译器目录下输入命令：

```
TLIB LIB\GRAPHICS.LIB+EGAVGA
```

TLIB 命令的意思是将 EGAVGA.OBJ 的目标模块装到 GRAPHICS.LIB 库文件中。

（3）在程序中 initgraph()函数调用之前加上一句：

```
registerbgidriver(EGAVGA_driver);
```

该函数告诉连接程序在连接时把 EGAVGA 的驱动程序装入到用户的执行程序中。

经过上面 3 个步骤，编译链接后的执行程序可在任何目录或其他计算机上运行。以下程序是在假设已作了前两个步骤的前提下编写的：

```
#include<stdio.h>
#include<graphics.h>
int main(void) {
    int gdriver=DETECT,gmode;
    registerbgidriver(EGAVGA_driver);// 把图形驱动程序装入到执行程序中
    initgraph( &gdriver, &gmode,"c:\\tc");
    line(100,100,200,200);              //画线
    getch();
```

```
        closegraph();
        return 0;
    }
```

这下终于可以独立运行了。如果想初始化成其他图形分辨率，比如 CGA 分辨率，则只需要将上述步骤中有 EGAVGA 的地方用 CGA 代替即可。事实上，可以将所有 BGI 文件一劳永逸地通过步骤 1，2 装入 GRAPHICS.LIB 库文件中。

2.5　赛车的完整图画

以下是一个用线和矩形绘制的简单赛车 bus1.c：

```
#include <stdio.h>
#include <graphics.h>
void main(void) {
    int gdriver = DETECT , gmode;
      // 初始化图形模式和逻辑变量
    initgraph(&gdriver, &gmode, "");
    setbkcolor(7);//设置背景颜色，灰色
    setwritemode(XOR_PUT);//设置写模式，异或
    setcolor(BLUE);//设置前景颜色，蓝色
    setlinestyle(SOLID_LINE,0,3);//设置线形，实线，线宽 3
      //画小汽车的轮廓
    rectangle(280,350,320,390);//矩形绘制函数
    rectangle(270,340,330,350);
    rectangle(290,320,310,340);
    rectangle(270,390,330,400);
    setcolor(5);//设置前景颜色，紫色
      //画小气车的车干
    line(290,350,290,390);//画线函数
    line(300,300,300,320);
    line(300,350,300,390);
    line(310,350,310,390);
    line(285,300,315,300);
    getch();
    closegraph();
}
```

赛车的图像如图 2-1 所示。

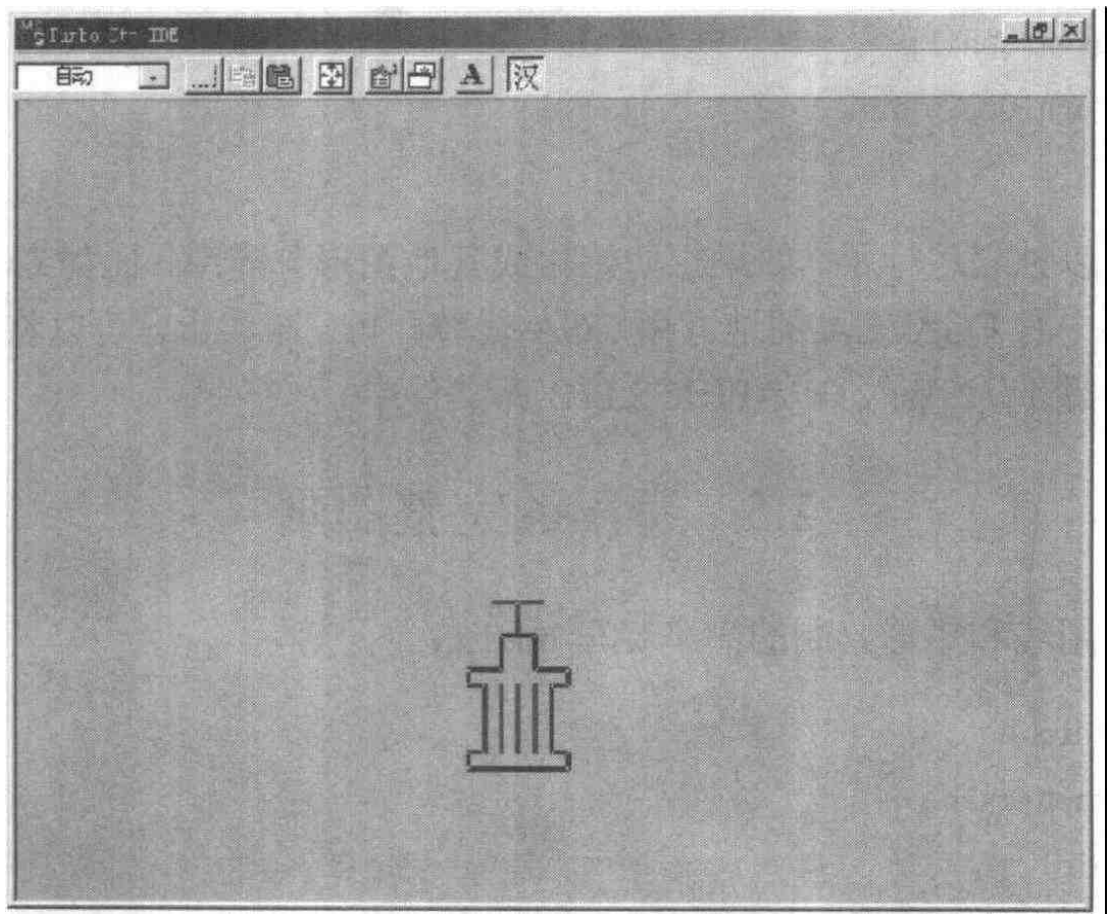


图 2-1　赛车图像

完整程序在赛车基础上绘制了赛道、周围的绿化树木和简单集装箱车 bus2.c 请查阅所附光盘的“source\2”目录。

完整的赛车图像场景如图 2-2 所示。

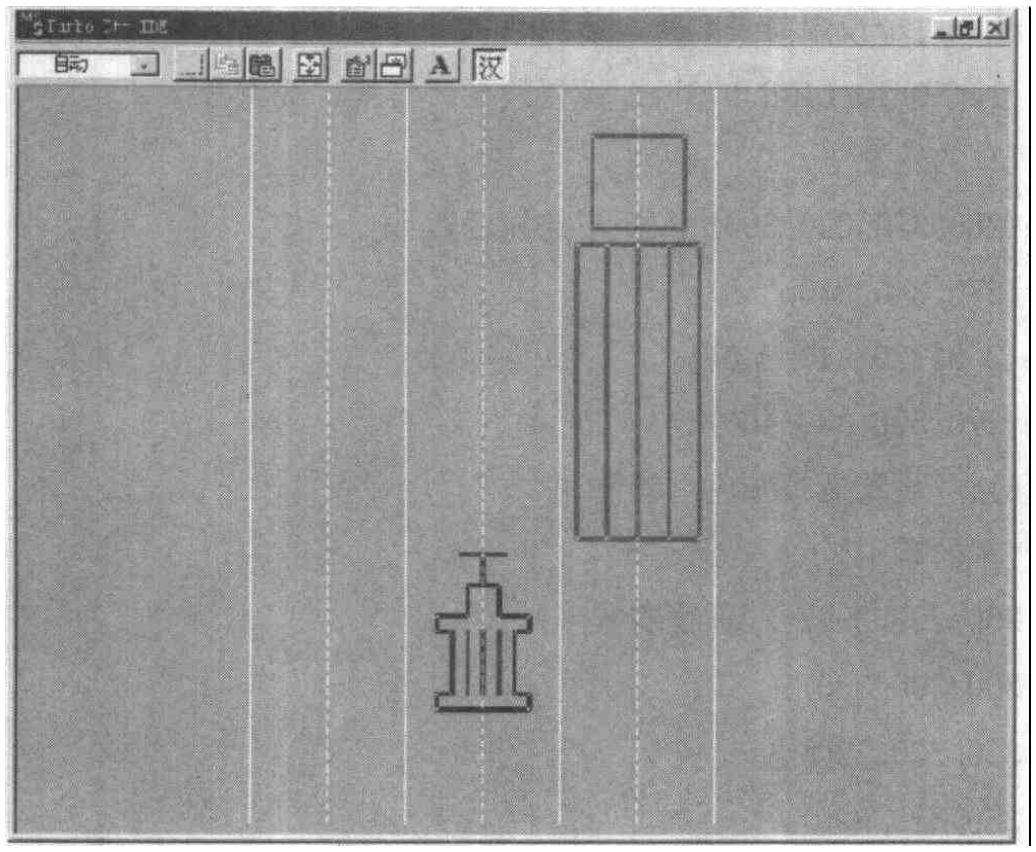


图 2-2　赛车场景

2.6　本 章 小 结

通过本章读者可以自如地设计任意的二维图形或在图形模式下写字。本章的最大难点是如何利用 initgraph()函数成功设置图形模式，这里最关键的是图形驱动文件路径要设置正确。例如下面设置图形模式语句：

initgraph(&gdriver, &gmode. "c:\\tc");

其中 gdriver 变量表示图形驱动器，我们将其定义为自动监测方式（赋值为 DETECT）；"c:\\tc"表示*.bgi 图形驱动文件在 C:\tc 目录下，并且必须要写成双斜杠。

当我们将自己设计的图形程序编译成 exe 文件后，发现在其它目录下无法使用。原因是图形驱动程序并没有包含在 exe 文件中。这时候我们必须建立独立的图形程序。其过程为：

（1）将驱动程序 EGAVGA.BGI 转换成 EGAVGA.OBJ 的目标文件；

（2）将 EGAVGA.OBJ 的目标模块装到 GRAPHICS.LIB 库文件中；

（3）在调用 initgraph()函数之前先调用 registerbgidriver()函数。

第3章　简 单 动 画

本章导读

上一章用C语言实现的图形绘制，然而这并不能满足我们的需要。因为除了最愚蠢的智力问题游戏可以使用图片形式以外，几乎所有的游戏都要用到动画。

事实上，游戏在屏幕中最直接的体现便是各类动画。学会了用C语言图形函数画图之后，可以做出各类我们希望的图形。然而这些画面都是静止不动的。如何让它们动起来，实现我们进一步希望的动画效果呢？

当时我认为从图形到动画的关键有两方面：

（1）可以通过保存图形、改变对各类图形坐标变量和重新显示图形的方法实现图形位置或者形状变化；

（2）C语言是否有延迟函数，如果没有那么就进行计数循环。

对于第二点我找到了一个时间延迟函数delay()（查阅所附光盘的“book\附录F”），这是实现动画的最基本思路，本章则将具体地讨论如何实现简单的动画。

本章重点

（1）实现动画最普通的思路：幻灯片思路和局部重画思路；

（2）熟练运用图形保存和图形恢复函数来实现图形在屏幕上的重画；

（3）利用C语言标准函数库提供的异或算法来实现简单动画。

3.1　实现动画思路

我们从小就看动画片，知道动画片的原理是将一幅一幅的图片排列起来，至少以每秒钟24幅的速度连续拍摄。这样一来骗过了我们迟钝的眼睛，使我们误以为看到的一切在运动，而忘记了这一切都只是静止图片组成的。

这一思路对我们做电脑动画非常有帮助。它很容易让我们联想到将屏幕作为一张图片，每次对屏幕这样的图片进行重新绘制。具体思路如下：

（1）在屏幕上画一个要运动的图像（比如用circle()函数画一个圆）；

（2）停留一些时间（事实上非常短，很可能只有几十到几百毫秒）；

（3）清除整屏；

（4）在刚才画球位置相近处（固定增量）重新画那个圆；

（5）反复2~4步骤。

见如下图形3-1、3-2及图3-3。

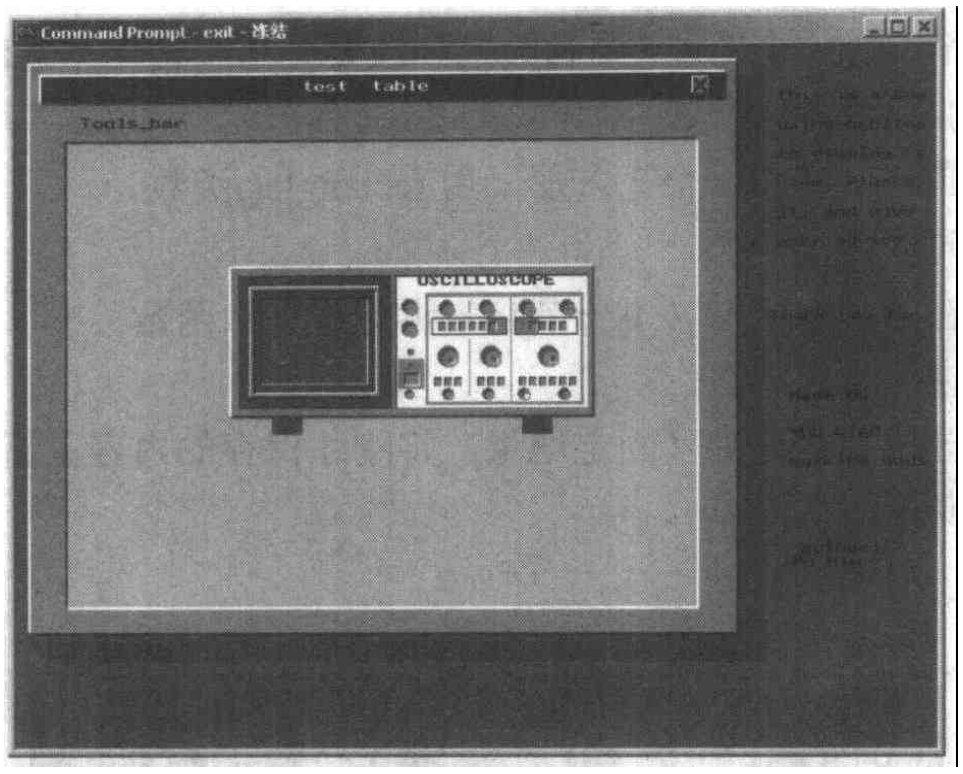


图 3-1

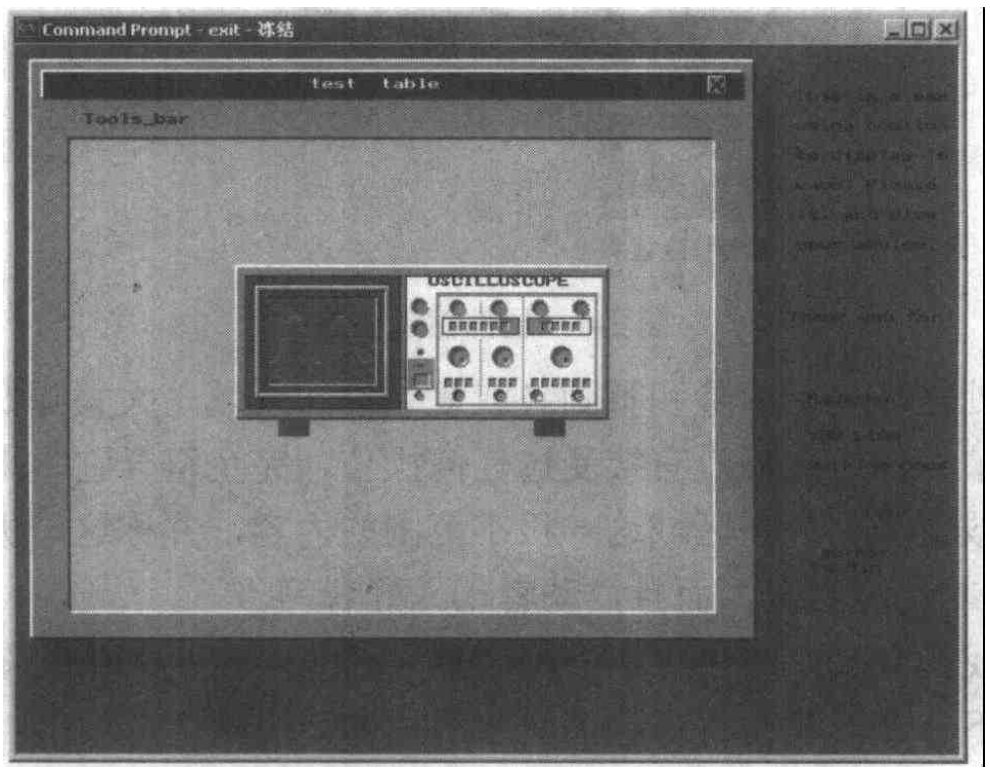


图 3-2

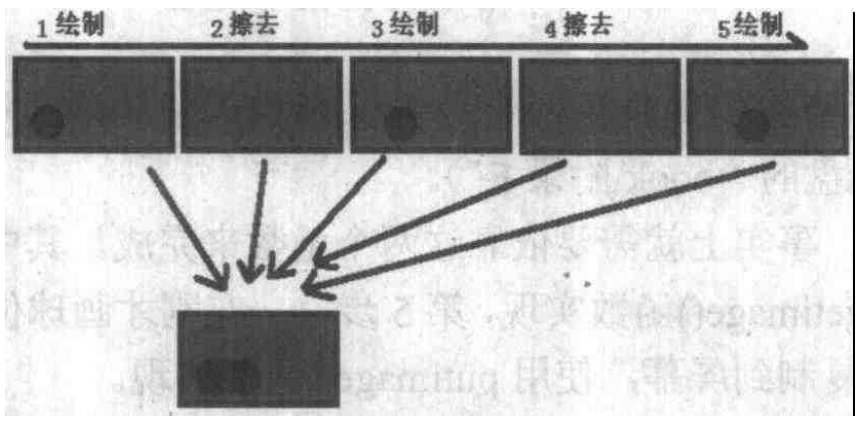


图 3-3　实现动画式幻灯片

我将这一思路赋予“幻灯片（或者动画片）”的名字。不过，开始有这样一个担心，动画片以每秒 24 幅的速度播放才逃过了我们的眼睛。而要清除、重画一个屏幕不知道要多少时间，而且要画的东西越多速度一定越慢，这怎么办呢？

刚才我们画的是一个圆在运动。那么如果它边上还放了一个不运动的正方形。如果还

是用刚才那 5 步思路，我们每次除了要画圆之外还要画那个根本不运动的正方形，这在时间方面是非常浪费的。我们考虑改进刚才的动画思路：

（1）在屏幕上画一个圆和一个正方形（用 rectangle()函数）；

（2）停留一些时间；

（3）保存圆所在的那块区域（下一节将告诉你用什么函数）；

（4）清除圆所在区域内容；

（5）在刚才画球位置相近处（固定增量）将刚才保存的内容重新复制到屏幕；

（6）反复 2-5 步骤。

最后见图 3-4。

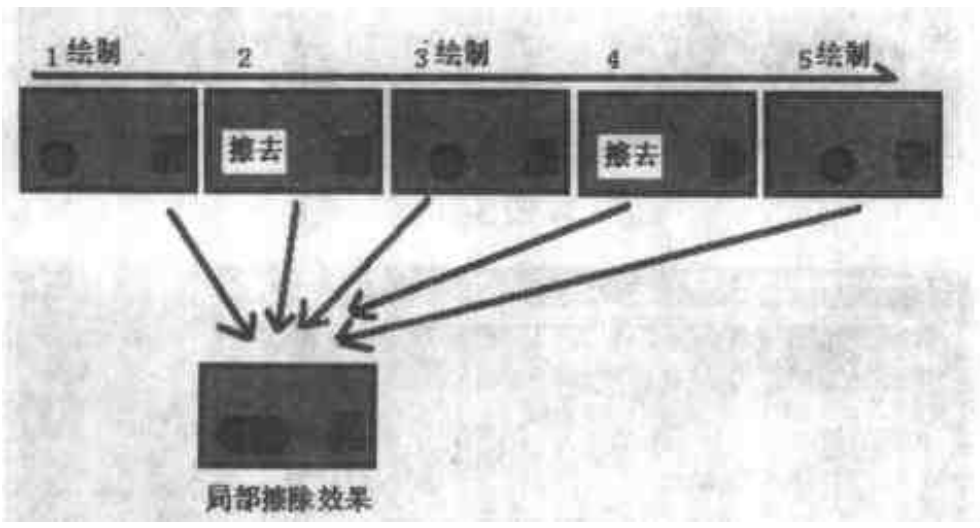


图 3-4　局部擦除图像

这一方法的好处是减少了清除和重画（或复制）的区域，大大降低了重复劳动。事实上，那个长方形在最初画好之后没有接受过任何操作，完全符合它本来在动画中的情况——静止。而我们则改变了圆所在区域这一极小范围的内容，并称这一优化思路为“局部重画”。此思路主要依靠动静结合，而且，动者动、静者静。

现在的问题是我们到底如何保存和重画屏幕区域的内容，C 语言图形函数库提供了怎样的对应函数？在下一节，我们将对这个问题进行详细的阐述。

3.2　屏幕保存与恢复

C 语言图形函数库给我们提供了两个用于保存和重画屏幕区域的函数 getimage()和 putimage()（查阅所附光盘的“book\附录 F”）。

上一节的优化思路，事实上就需要依靠这两个函数来完成。其中，第 3 步——保存圆所在的那块区域，使用 getimage()函数实现，第 5 步——在刚才画球位置相近处（固定增量）将刚才保存的内容重新复制到屏幕，使用 putimage()函数实现。

这里值得一提的是两个函数中 bitmap 指针参数指向为保存屏幕区域申请的内存空间。内存空间的大小事实上就是从左上角点(left,top)到右下角点(right,bottom)那块区域的大小。C 语言图形函数库为我们提供了一个计算机图像大小的函数 imagesize()（查阅所附光盘的“book\附录 F”）。

在得到图像大小之后我们可以通过内存分配函数 malloc()或者 farmalloc()来申请内存空间。请注意不要申请太大的内存空间，否则很可能返回 NULL。事实上我们经常会碰到游戏中要求保存当前画面（整个屏幕）的情况，而用 malloc()去申请屏幕大小的内存是一定无

法申请到的。

如何解决这个内存瓶颈的问题呢？我们考虑使用将屏幕四分保存到文件，然后文件四分恢复到屏幕的方法，因为四分之一屏幕大小的内存还是可以申请到的。以下是具体实现的例程 demo.c：

```
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
void save_all(char *filename) {//保存全屏子函数
    FILE *fp;
    unsigned long size = imagesize(0,0,639,120);//设置屏幕1/4大小单位保存尺寸
    unsigned long save_l;
    int save_i;
    char *buffer;
    fp = fopen(filename,"wb");//新建或者打开图形数据文件
    buffer = (char *)malloc(size);//申请1/4屏幕大小空间
    for(save_i=0;save_i<4;save_i++) { //分4次保存屏幕
        save_l=38726L*save_i;//计算文件内偏移
        fseek(fp,save_l,SEEK_SET);//文件内偏移定位
        getimage(0,120*save_i,639,120*(save_i+1),buffer);//取屏幕内容到内存
        fwrite(buffer,size,1,fp);//将内存数据写入文件
    }
    free(buffer);//释放内存
    fclose(fp);
}

void load_all(char *filename) { //恢复全屏子函数
FILE *fp;
unsigned long size = imagesize(0,0,639,120);
unsigned long load_l;
int load_i;
char *buffer;

if ( (fp=fopen(filename,"rb"))==NULL ) { //打开已经保存了屏幕数据的文件
    setcolor(12);
    outtextxy(450,460,"<File not found!>");
    fclose(fp);
```

```
        return;
    } buffer = (char *)malloc(size);//申请 1/4 屏幕的内存

    for(load_i=0;load_i<4;load_i++) { //分 4 次恢复图像到屏幕
        load_l = 38726L*load_i;// 计算文件内偏移
        fseek(fp,load_l,SEEK_SET);// 文件内偏移定位
        fread(buffer,size,1,fp);//将文件内数据读取到内存
        putimage(0,120*load_i,buffer,COPY_PUT);//将内存数据显示到屏幕
    } free(buffer);//释放内存
    fclose(fp);
}

void main() {
    int driver=DETECT,mode=0,i;
    initgraph(&driver,&mode,"");//初始化图形模式
    randomize();//初始化随机数发生器
    for(i=0;i<100;i++) { //循环随机绘制图像
        setcolor(rand()%16);//随机设置前景颜色
        circle(rand()%getmaxx(),rand()%getmaxy(),rand()%100);//随机画圆
    } save_all("screen.dat");//保存屏幕画面到 screen.dat 文件
    getch();
    cleardevice();//清除屏幕
    getch();
    load_all("screen.dat");//将 screen.dat 内图形数据恢复到屏幕
    getch();
    closegraph();//关闭图形模式
}
```

3.3 重画动画实例

在第 3.1 节提到两个动画思路。第一个是每次全屏清除“幻灯片（或者动画片）”；第二个是每次只重画运动对象的“局部重画”。在第 3.2 节提到保存和恢复屏幕区域的函数，它可以帮助实现局部重画。

以下的两个程序，是分别使用第一和第二种思路来实现一个运动的圆和一个静止的正方形。

幻灯片思路例程 ball1.c：

```
#include <graphics.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <conio.h>
int main(void)
   int gdriver = DETECT, gmode;
   void *ball;
   int x, y,maxx,
   unsigned int size;
   initgraph(&gdriver, &gmode, "");
   maxx = getmaxx();
   x = 0;
   y = 200;
   rectangle(x, y+11, x+20, y-31);
   circle(x+10, y, 10);
   size = imagesize(x, y-10, x+20, y+10);
   ball = malloc(size);
   setfillstyle(SOLID_FILL, BLACK);
   while (!kbhit()) {
      cleardevice();  //擦除全屏
      x += 10;
      if (x >= maxx) {
     x - 0;
      rectangle(0,211,20,231); //重画正方形
      circle(x+10,y,10); //重画圆
      delay(100);
   } }

   free(ball);
   closegraph();
      return 0;
   }
```

局部重画思路例程 ball2.c：

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void) {
   int gdriver = DETECT, gmode;
   void *ball;
```

```
int x, y,maxx;
unsigned int size;
initgraph(&gdriver, &gmode, "");
maxx = getmaxx();
x = 0;
y = 200;
rectangle(x,y+11,x+20,y+31);
circle(x+10,y,10);
size = imagesize(x, y-10, x+20, y+10);
ball = malloc(size);
getimage(x, y-10, x+20, y-10, ball);
setfillstyle(SOLID_FILL, BLACK); //设置填充模式为：黑色实心填充
while (!kbhit()) {
    bar(x, y-10, x+20, y+10);      //用黑色填充圆所在区域，擦除部分区域
    x += 10;
    if (x >= maxx) {
    x = 0;
    putimage(x, y-10, ball, COPY_PUT); //将保存下来的圆重画到新位置
    delay(100);
} }
free(ball);
closegraph();
return 0;
}
```

在运行了两个程序之后，我们明显发现前一个程序在圆每次移动的时候，正方形发生明显的闪动；而在后一个程序中这个问题得到了彻底的解决，圆依然在运动，而正方形一点都没有闪动。

事实上这两个程序基本结构完全一致，只有在擦除和重画动画对象——圆的时候用了不同的语句。

在擦除的时候，ball1 使用 cleardevice()函数将全屏连同那个不动的正方形一起擦除；而 ball2 只是将 setfillstyle()和()函数组合使用来清除圆所在的很小区域中的图形，而不影响边上的正方形。

在重画的时候，ball1 重画了正方体（因为给擦掉了）和圆；ball2 只是用 putimage()函数将先前保存的圆重新复制到屏幕新的位置。

3.4 简单动画实现

逻辑运算中的“异或”指 A 与上 B 的非同 B 与上 A 的非进行或操作的结果，具体用

公式表示为：A 异或 B=A*$\overline{B}$+$\overline{A}$*B

异或逻辑运算有一个特点，如果 A 和 B 是相同的，那么 A 与 B 以获得结果一定为 0（1 异或 1=1*0+0*1=0+0=0）。或者说，自己异或自己等于没有进行任何操作。

很多人会问，异或与图形绘制有什么关系呢？

如果我们在一个位置画了一条红色的线，然后在这个位置再画一次结果是什么？当然是红线还在那里，没有变化。可是如果在画第一条线之前就设置用异或方式画线，那么当我们画第二条红线的时候奇迹便发生了——屏幕上那条红线消失了。现在让我们来试试看如下清单 line.c：

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void) {
    int gdriver = DETECT, gmode;
    initgraph(&gdriver, &gmode, "");//初始化图形模式
    setwritemode(XOR_PUT);//设置异或模式
    setcolor(RED);//设置前景颜色为红色
    line(100,200,500,200);//画红线
    getch();//取键盘按键
    line(100,200,500,200);//在原来的线的位置再画一条红线，红色的线消失了
    getch();
    closegraph();//关闭图形模式，恢复到文本模式
    return 0;
}
```

运行结果如图 3-5 和图 3-6 所示。

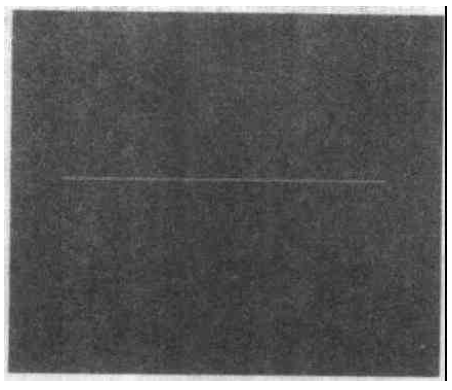


图 3-5　最初画面（第一次画红线后）

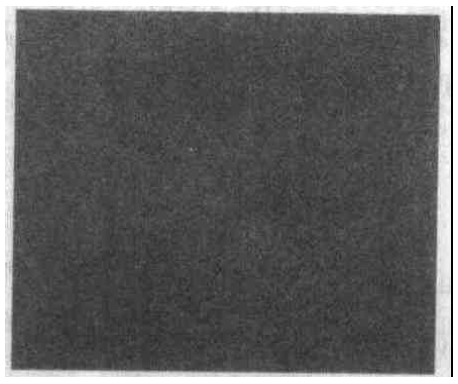


图 3-6　按键后（第二次画红线后）

在要进行异或操作之前，我们使用了一个设置异或模式的函数 setwritemode()（查阅附录 D）。

异或可以帮助我们用最简单的方法——二次完全重画方法擦除原先的图形。那么如何将它和动画联系起来呢？

这里在第 3.1 节两个动画思路的基础上提出第三个“异或思路”：

（1）设置异或模式，然后在屏幕上画一个圆和一个正方形；

（2）停留一些时间；

（3）在刚才画圆的位置用同样的颜色再画一个圆（圆消失了）；

（4）在刚才画圆位置相近处（固定增量）再画一个圆；

（5）反复2-4步骤。

以下是使用异或思路实现一个运动的圆和一个静止的正方形的例程ball3.c：

```
#include <graphics.h>
#include <stdlib.h>
#fnclude <stdio.h>
#include <conio.h>
int main(void) {
   int gdriver = DETECT, gmode;
   void *ball;
   int x, y,maxx;
   unsigned int size;
   initgraph(&gdriver, &gmode, "");
   maxx = getmaxx();
   x = 0;
   y = 200;
   rectangle(x,y+11,x+20,y+31);//绘制矩形
   circle(x+10,y,10);//绘制圆
   size = imagesize(x, y-10, x+20, y+10);//计算保存区域大小
   ball = malloc(size);//申请保存区域内存
   getimage(x, y-10, x+20, y+10, ball);//保存圆所在位置图形到内存
   while (!kbhit()) {
      putimage(x, y-10, ball, XOR_PUT);//用异或方式在原处绘制圆，圆消失
      x += 10;//计算圆移动增量
      if (x >= maxx) {
         x = 0;
         putimage(x, y-10, ball, XOR_PUT);// 用异或方式在新位置绘制圆，圆出现
         delay(100);//延迟一定时间
      } }

   free(ball);
   closegraph();
   return 0;
}
```

该程序的运行情况，和 3.3 节第二个例程的效果差不多。但从实际画点开销角度来看，3 个圆球运行的例程是不一样的。

3.3 节第一个例程使用清屏完全重画的方法，每次的画点开销为：

清屏+20*4 点（重画正方形）+10*4 点（重画圆形）

3.3 节第二个例程使用局部填充、重画的方法，每次画点开销为：

20*20 点（局部填充清除）+10*4 点（重画圆形）=440 点

本节例程使用 XOR 异或方法重画，每次画点开销为：

10*4 点（重画圆形清除）+10*4 点（重画圆形）=80 点

由此可见，第二个例程 440 点的开销大于第三个例程 80 点；而第一个例程由于每次重画正方形造成不必要闪动而根本没有参与比较意义（不要计算为 120 点）。异或在这里是最好的选择。

3.5　用异或实现赛车动画

以下是在上一章赛车静止画面基础上使用异或方式实现的赛车动画效果 bus3.c：

```
#include <graphics.h>
static int u=0;
static int x=1;
static int j=-21;
static int i;
void road(void) {
    int h;
    for(h=0;h<4;h++)
        line(150+h*100,0,150+h*100,472);
    for(h=0;h<3;h++) {
        setlinestyle(3,0,1);
        line(200+h*100,0,200+h*100,472);
        settextstyle(1,HORIZ_DIR,3);
    } }
void tree(void) {
    int w;
    int poly[14];
    setcolor(10);
    for (w=-3;w<3;w=w+2) { //绘制左边的树木
        line(85,-25+u*15+w*157,85,35+u*15+w*157);
        line(95,-25+u*15+w*157,95,35+u*15+w*157);
        line(105,-25+u*15+w*157,105,35+u*15+w*157);
        line(115,-25+u*15+w*157,115,35+u*15-w*157);
```

```
        line(75, -9+u*15+w*157, 75, 19+u*15+w*157);
        line(125, -9+u*15+w*157, 125, 19+u*15+w*157);
    }
    for (w=-2;w<3;w=w+2) { //绘制右边的树木
        poly[0] = 530;
        poly[1] = u*15+w*157;
        poly[2] =515;
        poly[3] = 25+u*15+w*157;
        poly[4] =485,
        poly[5] =25+u*15+w*157 ;
        poly[6] =470;
        poly[7] =u*15+w*157 ;
        poly[8] =485;
        poly[9] =-25+u*15+w*157;
        poly[10] =515;
        poly[11] =-25+u*15+w*157 ;
        poly[12] = poly[0];
        poly[13] = poly[1];
        drawpoly(7,poly);//多边形函数
    } }
void bus1(void) {
    i=2;
    setlinestyle(SOLID_LINE,0,3);
    do { //进入动画循环
        if((x!=7)&&(x!=16)&&(x!=23)) setcolor(x);
        else     setcolor(2);
//以下绘制卡车
        rectangle(170+i*100, j*10, 230+i*100, 60+j*10);
        rectangle(160+i*100, 70+j*10, 240+i*100, 260+j*10);
        line(180+i*100, 70+j*10, 180+i*100, 260+j*10);
        line(200+i*100, 70+j*10, 200+i*100, 260+j*10);
        line(220+i*100, 70+j*10, 220+i*100, 260+j*10);
        tree();//绘制树木
        delay(5);//延迟一定时间

        if((x!=7)&&(x!=16)&&(x!=23)) setcolor(x);
        else     setcolor(2);
//以下擦除卡车
```

```
        rectangle(170-i*100, j*10, 230+i*100, 60+j*10);
        rectangle(160+i*100, 70+j*10, 240+i*100, 260+j*10);
        line(180-i*100, 70+j*10, 180+i*100, 260+j*10);
        line(200-i*100, 70+j*10, 200+i*100, 260+j*10);
        line(220+i*100, 70+j*10, 220+i*100, 260+j*10);
        tree();//擦除树木
        u++;//树木起始位置偏移累加
        j++;//卡车起始位置偏移累加
    }while((j<=47)&&(u<=62));
}
void car(void) {
    setcolor(BLUE);
    setlinestyle(SOLID_LINE, 0, 3);
    rectangle(280, 350, 320, 390);
    rectangle(270, 340, 330, 350);
    rectangle(290, 320, 310, 340);
    rectangle(270, 390, 330, 400);
    setcolor(5);
    line(290, 350, 290, 390);
    line(300, 300, 300, 320);
    line(300, 350, 300, 390);
    line(310, 350, 310, 390);
    line(285, 300, 315, 300);
    setcolor(BLUE);
}
void main(void) {
    int gdriver = DETECT , gmode;
      // 初始化图形模式和逻辑变量
    initgraph(&gdriver, &gmode, "");
    setbkcolor(7);
    setcolor(WHITE);
    setwritemode(XOR_PUT);//设置异或模式
    road();//绘制路
    car();//绘制赛车
    bus1();//向后运动的大卡车和相对向后运动的树木，使用异或
    closegraph();
}
```

这里需要注明的是，不同机器在运行这个程序的时候动画速度不同，有的很快、有的很慢。原因是这里使用的 delay()延迟函数受 CPU 速度影响。目前要解决这个问题的办法只有大家自己动手改变 delay()里面的毫秒参数到适合的速度。在稍后的章节中将告诉大家如何得到精确的延迟时间。

3.6　本 章 小 结

动画就是画动，就是将一个个静止的画面通过时间排序展现出来所产生的视觉效果。

通过本章的学习我们可以将一个个独立的图形画面制作成动态画面。实现动画最常用方法是重画技术，而重画中也可分为幻灯片技术（全部重画）和局部重画技术。

重画技术的过程：

（1）绘制要进行动画的图形；

（2）延迟一定时间；

（3）擦去该图形（可以全屏擦去，也可以只将动画图形擦去）；

（4）在下一个递增位置重画此图形。

C 语言图形函数库中提供了一个保存屏幕画面的函数 getimage()和一个恢复保存画面到屏幕的函数 putimage()，再加上时间延迟函数 delay()就可以实现动画了。

另一个实现动画的常用技术就是异或技术。异或技术的基本原理：A 与上 B 的非同 B 与上 A 的非为 0。在图形绘制中使用异或模式两次绘制同样颜色的相同图形，图形就会消失。异或技术实现动画的过程：

（1）绘制要进行动画的图形；

（2）延迟一定时间；

（3）在同一位置用同样颜色和异或模式重画该图形（动画图形消失）；

（4）在下一个递增位置重画此图形。

学后建议

（1）使用重画和异或动画技术让前一张图形动起来；

（2）无论重画和异或实现的动画闪动都很厉害，考虑如何进一步优化动画技术和提高代码效率，从而减少闪动；

（3）进一步考虑 C 语言实现动画的其它技术（建议参看第 9 章中动画实现办法分类）。

第 4 章　简单图形游戏

本章导读

终于可以开始尝试图形游戏了。此时，脑海中浮现出俄罗斯方块的游戏情景，这让我们异常兴奋。

记得大学的时候我的一位老师曾经说过，如果较为全面的掌握了基础游戏编程的各方面知识，计算机的学习可以算是达到一定层面了。他的话中包含的另一层意思是，我们如果要学习游戏编程必须有一定的计算几个方面课程知识。比如刚才说到的俄罗斯方块游戏，如果仅仅是如同前两章实现图形或者简单动画那并不复杂，问题是在游戏中要让方块变换、堆积和消层这就必然涉及数据结构（这里主要体现为数组的应用）和算法的许多知识。

结合我的一些经验和教训，希望大家能够在开始游戏设计前先打下一些计算机其它课程知识的基础，数据结构和算法两门课程尤其重要。

当然对于本章这些还只能算是题外话，我们将着重强调从动画到游戏的实现方法。

本章重点

（1）掌握游戏流程的四大要素：动画（这里指重画和延迟，不包括循环）、响应、运算、循环；

（2）游戏中键盘响应问题，在游戏中使用 kbhit()或者 bioskey()函数不仅可以响应键盘输入，还可以保证在没有键盘触发时游戏动画循环继续运行下去，而不是如同 getch()函数直到有键盘触发才能运行下一个指令；

（3）随机数实现办法，随机数是游戏制作中常用和必备的一样东西，也是人工智能最基本的要素之一。

4.1　从动画到游戏

动画和游戏从编程设计者角度来说最大的区别是，动画是设计者早已安排好的全部变换过程（仅指画面）；而在游戏中加入了响应，设计者给出的只是一些规则和与这些规则对应的部分变换过程（包括画面和内部变量）。

再从游戏者角度来看动画和游戏两者的区别。显然对于动画我们除了笑就没有什么可以参与的了；而游戏的主角却是它们自己，我们的一举一动（实际就是按键盘和鼠标等操作）将决定游戏的发展方向。也就是说一个是没有交互，一个是有交互。

的确，动画要比游戏简单了很多。让我们再回忆一下上一章的内容，好像我们说到的一切都只是如何实现动画效果。而在游戏中我们除了要考虑动画，还要考虑响应游戏者（用户输入），并且要考虑针对用户输入进行相应的操作。见流程图 4-1。

看来，从动画到游戏最关键的一步就是解决用户输入问题。那么再让我们来仔细想想上一章中动画程序的通用流程结构。

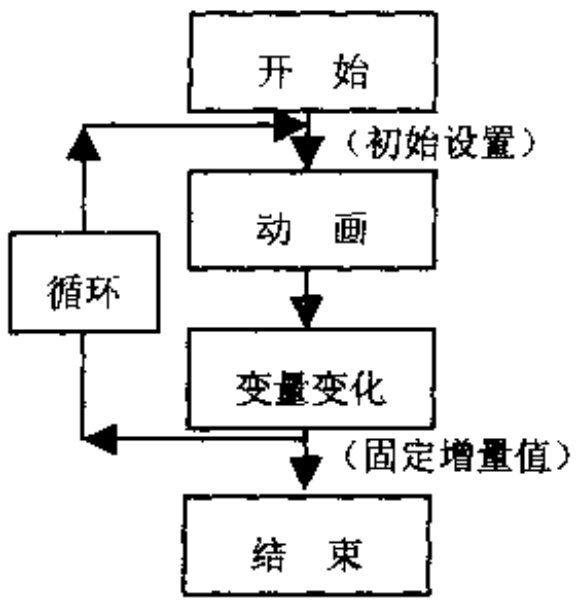


图 4-1　动画实现过程示意

而从动画到游戏我们只再流程中增加了一个环节和改变了一个环节：

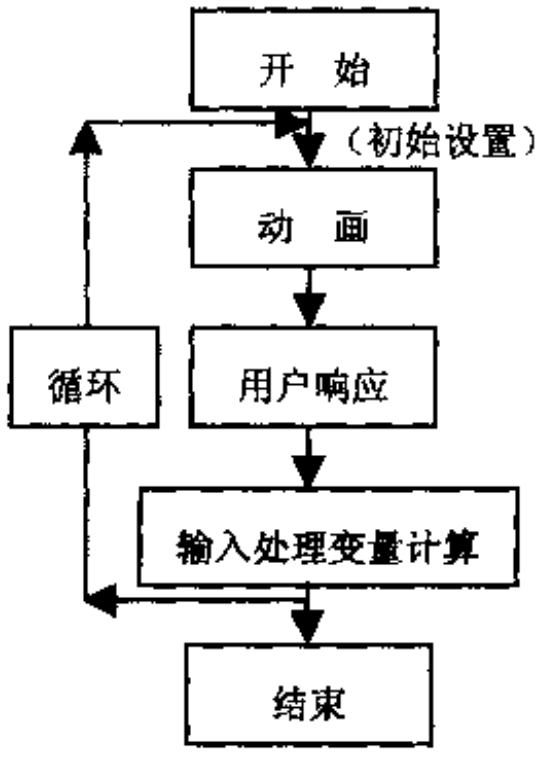


图 4-2　游戏实现流程示意

从程序角度来说，动画程序和游戏程序的主体都是一个 while()循环语句。动画程序的 while()循环语句的表达式通常只是退出条件；而游戏程序中 while()表达式则要加上检测用户输入的内容，并在循环内进行分类处理（比如用 switch … case 语句）。

从动画到游戏就只有一大步——用户输入，和一小步——输入处理。

4.2　简单用户响应

在玩游戏的时候，通常会碰到不想再继续玩下去的情况，随时可以按键盘上的"q"。游戏会问你：quit game?(y or n)，你按键盘 y，于是游戏结束。

这个部分一定是用键盘输入函数来实现的。我们很快会联想到 getch()函数。让我来试试看 quit1.c：

```
#include <conio.h>
void main(void) {
   int i;
```

```
while(1){
    printf("\nPress 'q' to quit:");
    i=getch();//读取键盘按键
    if(i--'q') ;//判断按键是否为"q"
        printf("\n quit game?(y or n)");
        i=getch();//读取键盘按键
        if(i=='y')//判断按键是否为"y"
            exit(0);
    } } }
```

可是在游戏过程中，我们要保持游戏主体的的进行这个程序就无法做到了。因为 getch()函数始终是处于等待键盘输入状态，只到有键盘输入的时候才执行下一个语句。

而现在需要的是在游戏主程序中检测是否有按下 q，没有的话继续向下执行主程序，相反就询问：quit game?(y or n)。

事实上 C 语言为我们提供了一个非常简单的检测键盘输入的函数 kbhit()（查阅所附光盘的“book\附录 F”），该函数的好处就是它不影响程序的运行，即使没有按键程序还是可以继续运行下去。如果使用一个 while 循环语句，将 kbhit()作为表达式，如以下语句：

```
while(!kbhit());
```

那么 while 语句将不断地循环，直到有按键发生才跳出。这是 getch()函数根本无法做到的。

如果将 kbhit()和 getch()函数联合使用不仅可以检测按键，还可以检测出键值。具体是通过 kbhit()函数检测按键，然后依靠 getch()函数将刚才的具体按键值读出，使用语句如下：

```
while(!kbhit());
i=getch();
printf("%d",i);
```

现在用 kbhit()函数来改写上面的游戏退出部分程序 quit2.c：

```
#include <conio.h>
void main(void) {
int i;
while(1) {
    printf("\nPress 'q' to quit:");
    while(kbhit()==0) {//判断是否有按键
        i++; }//此处中可放游戏主体
    i=getch();//读取按键
    if(i=='q') {//判断按键是否为"q"
        printf("\n quit game?(y or n)");
        i=getch();
        if(i=='y') exit(0);
```

```
    } } }
```

在 i++处可以替换成游戏主体、动画或者任意你想加入的内容。这是前一个使用 getch()函数的程序根本无法做到的。

kbhit()函数给我们带来了在不断循环的游戏中随时接受用户键盘信息的可能。

4.3　接收用户信息

kbhit()函数可以帮助我们检测是否有来自键盘的信息。kbhit()加上一个 getch()函数可以帮助我们检测来自键盘的具体键值。

可是，以上只是针对普通键而言的。如果要碰到来自键盘的功能键（如 4 个方向箭头），kbhit()加上 getch()是无法识别的。更何况它还是需要两个函数才能最终解决检测按键和读入按键的任务。

事实上，getch()函数返回值中低位为 ASCII 码，高位为扫描码。如果碰到普通键检测，只需要调用一次 getch()函数，则返回该键的 ASCII 码；如果碰到功能键则要调用两次 getch()函数，第一个 getch()函数返回低位的 0，第二个 getch()函数返回高位存放的扫描码。

于是，要检测普通按键和功能按键可以通过以下程序实现 key.c：

```
#include <conio.h>
void main(void) {
    int i;
    while(kbhit()==0);//判断是否有按键，如果没有则循环检测
    i=getch();//读取该键
    if(!i) {//如果低位为 0 则读取高位
        i=getch();//读取高位
        printf("%d",i);//显示高位扫描码
    } else printf("%c",i);//普通键显示低位键值
}
```

当碰到普通码的时候，输出刚才键入的字符；当碰到功能键的时候，输出功能键的扫描码。对于功能键这里使用到了 kbhit()和 2 个 getch()函数，够麻烦的了。然而谈到键盘上的 Shift、Ctrl、Alt、Insert、CapsLock 和 NumLock 这些特殊键，getch()函数也无能为力了。

C 语言为解决这个问题，提供了一个键盘接口函数 bioskey()（查阅附录 D）。

比如在俄罗斯方块游戏中，bioskey()函数可以使用参数 1 极其方便地检测左右移动和加速向下的方向按键；在玛莉游戏，bioskey()函数可以使用参数 1 和 2 配合轻易地检测 Ctrl 加方向键的加速跑运动（当然同时按键问题将在第 14 章接口中介绍）。

bioskey()函数在游戏中的一般使用方法通常是：

```
while (//没有退出游戏按键或者没有游戏结束标志) {
    while (bioskey(1) == 0) {
        //游戏主体
```

```
        } key = bioskey(0);
        //输入处理
    }
```

以下的程序检测键盘上任意一个按键，并显示它的 ASCII 码 bioskey.c：

```
#include <stdio.h>
#include <bios.h>
#include <ctype.h>
#define Right   0x01
#define Left    0x02
#define Ctrl    0x04
#define Alt     0x08
#define ScrollLock  0x10
#define NumLock   0x20
#define CapsLock   0x40
#define Insert    0x80
int main(void) {
   int key, modifiers;
   while (bioskey(1) == 0);//循环读取键盘按键直到有按键产生
   key = bioskey(0);//读取按键的 ASCII 码
   modifiers = bioskey(2);//读取 shift key
   if (modifiers) {//如果是 shift key，则判断具体的按键
      printf("[");
      if (modifiers & Right) printf("Right Shift");
      if (modifiers & Left)  printf("Left Shift");
      if (modifiers & Ctrl)  printf("Ctrl");
      if (modifiers & Alt)   printf("Alt");
      if (modifiers & ScrollLock) printf("ScrollLock");
      if (modifiers & NumLock)  printf("NumLock");
      if (modifiers & CapsLock)  printf("CapsLock");
      if (modifiers & Insert)   printf("Insert");
      printf("]");
   }
   if (isalnum(key & 0xFF))//如果是普通按键
      printf("'%c'\n", key);//显示键值
   else printf("%#02x\n", key);
   return 0;
}
```

俄罗斯方块游戏的相关画面如图 4-3、4-4 和图 4-5 所示。

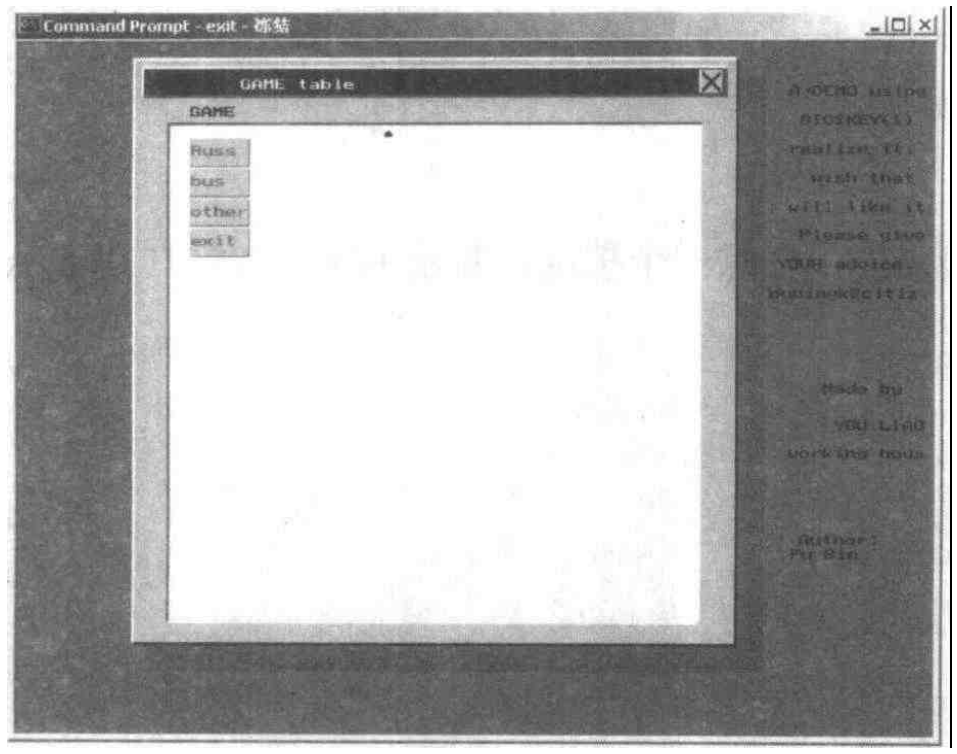


图 4-3　游戏开始菜单

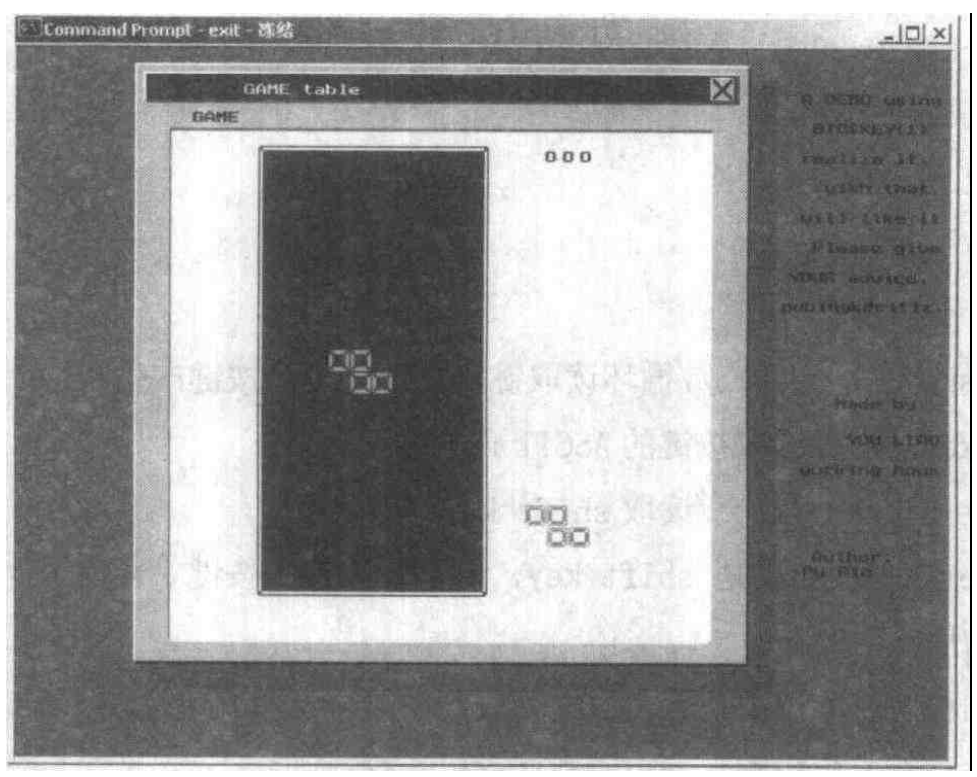


图 4-4　游戏开始后

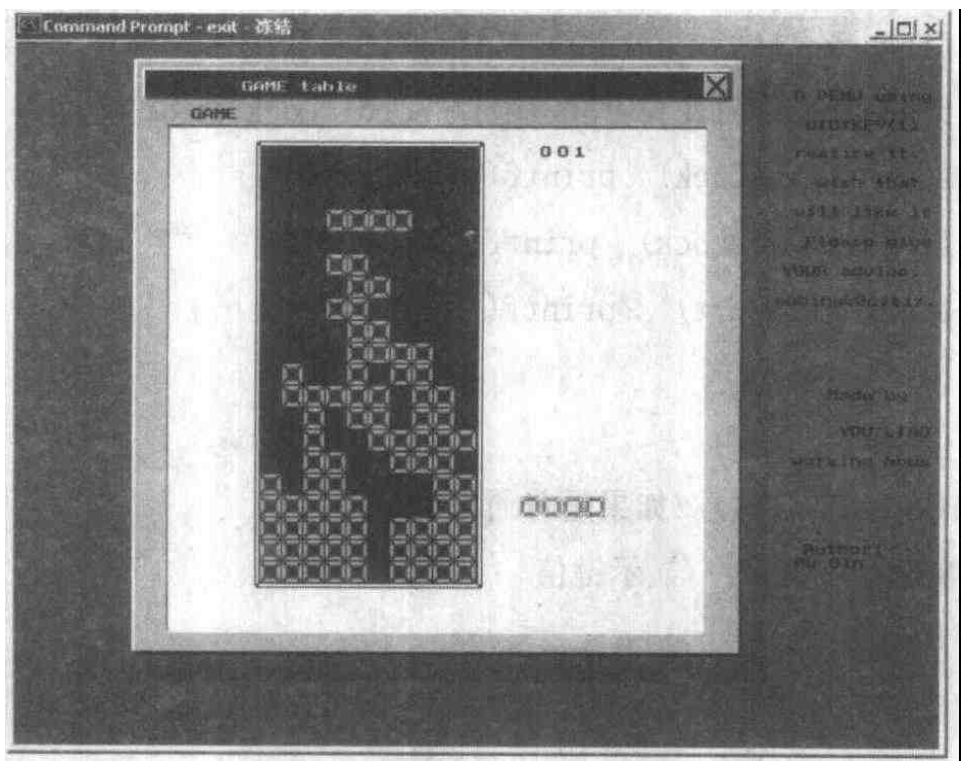


图 4-5　游戏进行中

表 4-1 给出了各类游戏中常用的功能按键的具体键值。

表 4-1　方向键及其扫描码

功能键名称	扫描码键值（10 进制，高位）	16 进制
左箭头	75	0x4b00
上箭头	73	0x4800
右箭头	77	0x4d00
下箭头	80	0x5000

4.4　配上其它东西

解决了用户输入的问题，一个游戏基本就可以实现了。不过，要使游戏更加生动一些就要加上更多的声音、动画以及使形式更加丰富。

4.4.1　配上声音

音效在游戏中是非常重要的。这里介绍用 PC 喇叭产生最简单的声音。在后面章节中将介绍如何驱动声卡使音箱发声。

C 语言函数库提供了两个声音函数 sound()、nosound()和延迟函数 delay()（查阅附录 D）。

sound()函数中频率参数的单位是赫兹（Hz），根据测试，调用参数 18Hz 以上的 sound()函数可以被人耳识别。以下是一个利用循环来改变声音频率的例子，发出的声音屏率从低到高 sound.c：

```
#include <dos.h>
void main(void) {
    int i;
    for(i=0;i<1000;i++) {//声音频率逐渐提高
        sound(i);//发声
        delay(10);//延迟
        nosound();//关闭 PC 喇叭
    }
}
```

在游戏中 sound()函数通常在某一游戏动画图像画好后处于延迟的时候使用。这里使用 delay()延迟函数一举两得，即使游戏动画图像进行了一定时间的停留，又使声音进行了一定时间的播放。

这里要再次提醒大家的是，不同机器在使用 delay()延迟函数时延迟时间不同，这是因为受 CPU 速度影响。最终我们将不使用 delay()函数作为延迟函数，这里只是权宜之计。

4.4.2　加入片头和片尾

相关的片头和片尾画面如图 4-6、图 4-7 和图 4-8 所示。



图 4-6　游戏片头

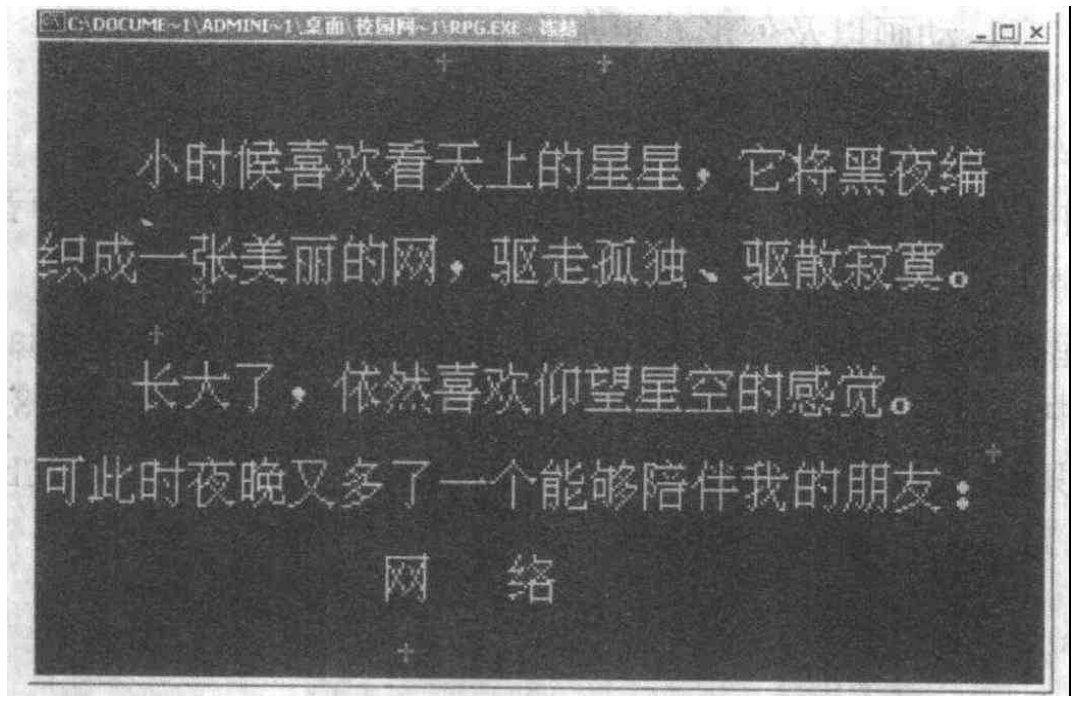


图 4-7　片头过渡

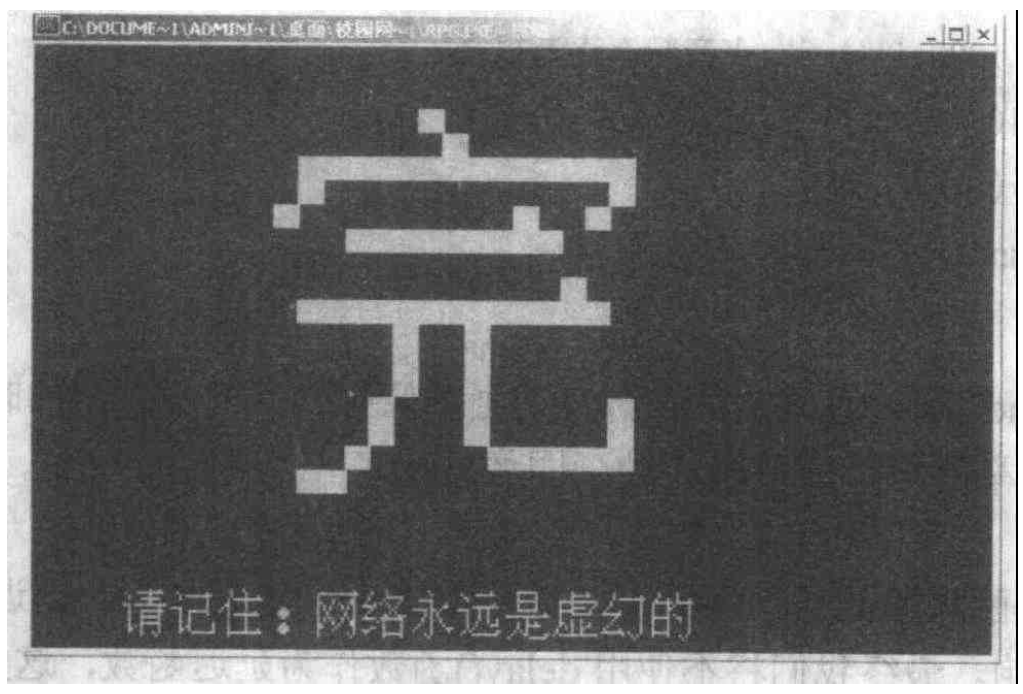


图 4-8　游戏片尾

增加一个片头动画或者一个游戏封面能够使游戏更加生动，也有利于介绍游戏的一些情况。这里可以利用上一章动画制作的知识尽可能地发挥自己的想象力做出吸引玩者的动画，不过不要忘记用 outtextxy()函数写上游戏的名称。

游戏片头不一定只有一个，如果需要对游戏进行详细的介绍。许多游戏在第一个片头结束之后通过按键进入一个帮助页面。这个页面中通常会说明游戏规则和游戏中使用到的按键，以及每个按键的作用。帮助页面一般适用于游戏规则较为复杂，操作较为麻烦的游戏。

在游戏结束的时候别忘了加上一个游戏制作说明。写上作者、制作时间、改进意见、下载位置和联系办法。

有了片头和片尾游戏看上去才会更加完整，当然最关键的还是要写出好的游戏来！

4.4.3　使用随机数

在游戏里常常会使用到随机数。比如，第 1 章“猜数字”游戏给出的被猜数最后就是用随机数产生的。而在游戏中更多的计算机控制对象的产生、行为也通常是根据随机数来控制的。

C 语言提供了随机数产生的函数 rand()（查阅所附光盘的“book\附录 F”）。

rand()和 random()函数都是随机数发生函数。rand()%num 功能相当于 random(num)函数，表示随机产生 0 到 num-1 中任意某个数字。以下是一个简单例程 rand1.c：

```
#include <stdlib.h>
#include <stdio.h>
void main(void) {
   int i;
   printf("rand()\n");
   for(i=0; i<3; i++)
      printf("%d\n", rand() % 100);
   printf("random()\n");
   for(i=0; i<3; i++)
      printf("%d\n", random(100));//取 0~99 中间的随机数显示
}
```

这两个函数都是随机数发生器，但是它们都只是调用已经固定的随机数源。于是，每次运行以上程序随机数产生的序列值都是一样的。

如何使每次运行同一程序的时候产生的随机数序列值不一样？C 函数库提供了一个依靠当前时间来初始化随机数发生器的函数 randomize()（查阅所附光盘的“book\附录 F”）。

每次在随机数发生器函数之前使用 randomize()函数，将改变随机数发生器源，从而使随机数不会出现任何重复现象。以下是一个随机数产生的完整程序 rand2.c：

```
#include <stdlib.h>
#include <stdio.h>
void main(void) {
   int i;
   randomize();//随机数发生器
   for(i=0; i<3; i++)
```

```
        printf("%d\n", rand() % 100);//取 0~99 中间的随机数显示
    }
```

现在运行这个程序，基本上每次产生的随机数序列都是不一样的。

4.5　赛 车 游 戏

在上一章“赛车”动画的基础上，结合用户键盘输入响应，做了一个简单的“赛车”游戏 bus4.c 请查阅所附光盘的“source\4”目录。

游戏运行图像序列见图 4-9 和图 4-10。



图 4-9　游戏开始画面后（封面：一个跳动的小球）

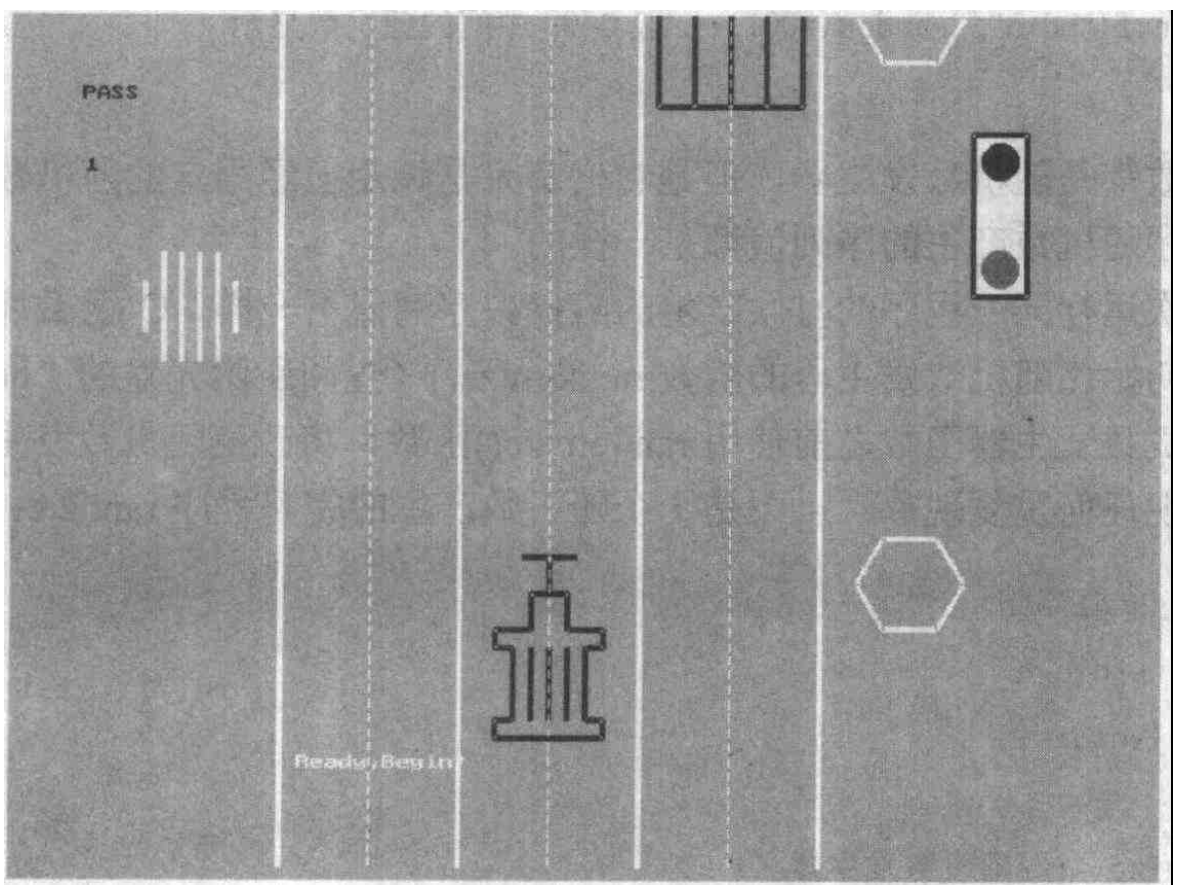


图 4-10　游戏中的小车

使用异或技术写的俄罗斯方块游戏 brick.c 请查阅所附光盘的“source\4”目录。

4.6　本 章 小 结

游戏从某种角度可以看作交互的动画，也就是说在动画的基础上只要加上用户响应和响应后的计算就可以实现简单的游戏。

在游戏响应问题上我们必须解决两个问题：

（1）保证实时地读取来自用户的键盘信息；

（2）如果没有用户键盘触发继续游戏。

我们无法单单使用最常用的 getch()函数来解决这两个问题，这时候必须考虑使用 kbhit()和 bioskey()函数。

通过本章我们还掌握了游戏制作中的一些辅助内容，比如声音的调用和随机数的使用。实现声音至少需要 sound()、nosound()和 delay()三个函数；而要产生真正的随机数至少需要 random()（或者 rand()）和 randomize()函数。

学后建议

（1）考虑进一步提高游戏响应的实时性问题，也就是如何保证用户按键不被遗漏并且被及时的读取，以及忽略过多的重复按键；

（2）以提高游戏的吸引力考虑为游戏配上哪些东西，并且如何实现它们。

第5章　图 形 模 式

本章导读

从这一章开始我们将抛弃 TC 自己的标准图形函数库（graphics.lib，标准图形头文件graphics.h），所有的图形相关函数将由我们自己制作。之所以不使用 TC 图形函数库的原因是，TC 标准函数通常强调更高的通用性，于是必然付出函数效率很低的代价，然而在游戏中最强调的就是代码高效性。我曾经使用 TC 标准图形函数库写了一个俄罗斯方块程序brick.c（配套光盘中有），方块在下落过程中的强烈闪动让我下定决心放弃了这个标准图形函数库。

事实上所有的 TC 标准函数库也都是用 TC 或者汇编语言写出来的，所以我们也完全可以写出功能相当、效率更高的函数来。只是我们写的函数在通用性上并不很高，但针对我们的游戏，通用性并不重要。

然而要真的靠自己去写类似于 TC 标准函数库中的函数要求我们在相关方面（比如硬件知识、基本代码运算效率等）都要有较扎实的基础。

第 2 章中我们介绍过图形模式初始化函数 initgraph()，通过它我们最高能够初始化到640*400 像素 16 色的图形模式，如此少的颜色选择明显无法满足我们的需要。本章我们将通过对显示适配器、显示模式和显示过程的更深入了解来完成图形模式初始函数。

本章重点

（1）了解计算机常用的显示模式，掌握 C 语言是如何利用 BIOS 中断来设置图形模式的；

（2）深入理解和掌握图形模式 13h 的工作模式、特点和显存寻址方式。

5.1　显示适配器与显示模式

显示适配器如同声卡一样，也是组成计算机的一个硬件部分。事实上显示器显示的所有内容都是计算机经过显示适配器处理后才传输到屏幕上去的。也就是说没有显示适配器就无法实现显示器的显示。

显示模式是显示适配器选择处理显示内容的方式。一个显示适配器可以在不同情况下选择不同显示模式，但是，显示模式也是受到显示适配器硬件条件限制的。例如，单色适配器不可能拥有图形显示模式。另一方面，不同的显示适配器也可能有相同的显示模式。

5.1.1　显示适配器

显示模式的设置是对显示卡的操作，而显示卡也就是显示适配器。目前通常使用的都是 PC 机，而从最初 IBM PC 机到现在，显示适配器经历了很长的一段发展。

最初的 PC 机将单色显示器和单色适配器（MDA）一起配置。这个时候还没有图形功能。随后的大力神单色显示适配器（HERCULES）提供了图形方式。最早的彩色图形来自

于 CGA（COLOR GRAPHICS ADAPTOR）适配器。之后才有了 EGA（ENHANCED GRAPHICS ADAPTOR）适配器和现在的 VGA（VIDEO GRAPHICS ADAPTOR）适配器。最后的 VGA 适配器终于提供了 256 色功能。

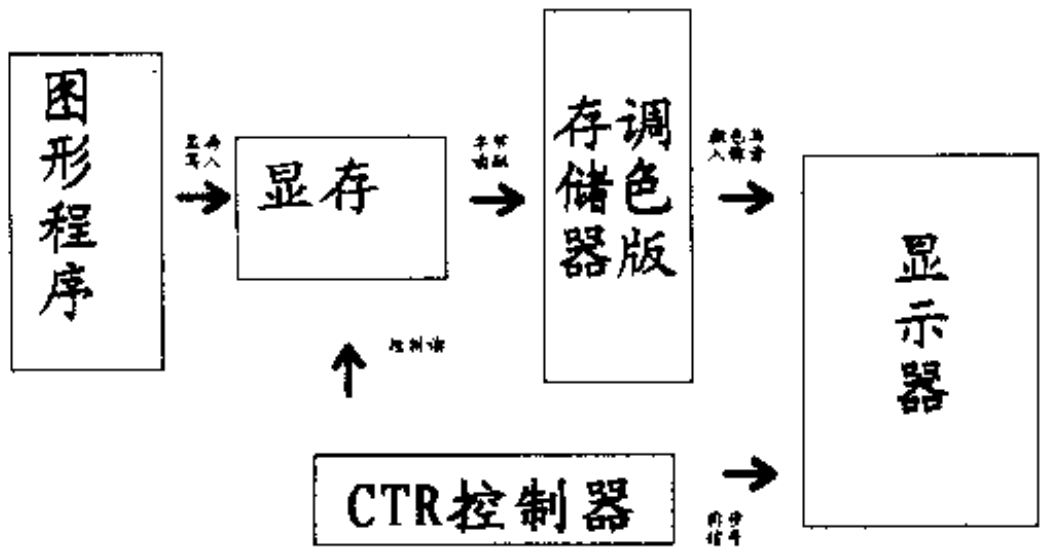


图 5-1　一般图形显示的过程

简单地说，计算机将需要输出的图像数据放入显示存储器，然后将每个显存地址中的数据和调色板寄存器进行比对后打到屏幕上。

5.1.2　显示模式

根据硬件的特点，每种适配器通常都提供了多种显示模式，尤其是后期适配器（如 VGA 适配器）。由于支持向上兼容，可以选择相当多的显示模式。有关标准 VGA 显示模式表请查阅所附光盘的“book\附录 F”。

从表中可以看到，文本模式的显存起始地址通常为 B8000H，而图形模式下显存地址通常是 A0000H。表中只有 VGA 适配器的模式 13H 提供了 256 色支持，而 640*480 在多种模式下出现，其分辨率最高。

我们写游戏可以选择自己喜欢的图形模式，这里建议使用模式 13H，毕竟 16 色太少了。

5.2　图形模式 13H

13H 是一个足以让我们激动的模式。多少经典游戏都是用这个模式写出来的。如今虽然 DOS 和 DOS 游戏走向末路，并且有更高分辨率和更多颜色支持的模式，但对于我们练兵目的而言这已经足够我们充分发挥了。

模式 13H 是 VGA 独有的模式。分辨率为 320*200,最大共存颜色为 256 种。由于它是线性映像显存的，给游戏编程带来了极大的方便（我们从前通过 initgraph()函数自检测初始化的图形模式通常是 640*480 的 16 色图形模式）。

由于 2^8=256，于是屏幕上的每个点（其实就是 256 个颜色中的一个）正好由显存中的一个字节来对应。320*200 个点正好由 320*200（64000）个字节表示。知道图形模式下显存的起始地址是 A0000H（文本模式为 B8000H），由于所有屏幕上的点都是按照从左到右、从上到下的顺序线性排列在显存中的，所以很容易得到屏幕上每个点在显存中的相应地址。

假设点的坐标为(x,y)，则其地址计算公式如下：

字节地址=A0000H+(y*320)+x

表 5-1 是一些特殊点的地址。

表 5-1　点及其对应地址

X	y	屏幕位置	地址
0	0	左上角	A0000H
319	0	右上角	A013FH
0	199	左下角	AF8C0H
319	199	右下角	AF9FFH
126	77	任意某个位置	A60BEH

图 5-2 是显存地址与屏幕的对应关系图，你可以将这张图看作 13H 模式下的屏幕显存。

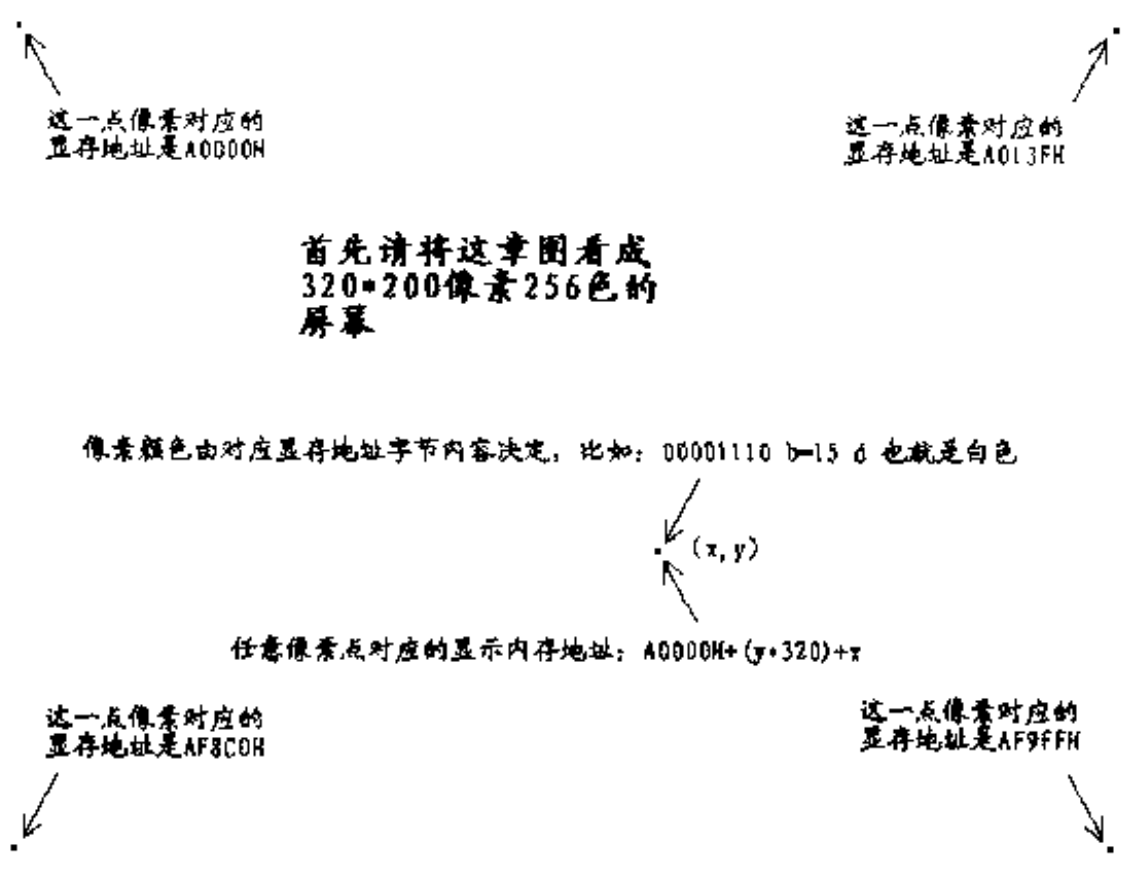


图 5-2　显存地址与屏幕点对应关系

从现在开始请不要再把屏幕当作一个完整的块面来看待了。换一个角度，将它看成一个有限个点组成的插板，而在这块插板上面的任何形状，无论是点、直线、圆还是填充图块，它们都是由一个个点通过某些规律组成的。我们只需要知道两样东西就能将任意美丽的画面展现在屏幕上，所要的两样东西便是：

（1）会向屏幕画一个点；

（2）懂得组成各类形状的算法。

很简单吧！在屏幕上画点的问题将在下一章中详细介绍，这里还是先来关注如何初始化图形模式。

要初始化模式 13H 可以通过 int86()函数触发 BIOS 中断 10H 视频服务中断或者汇编语言直接使用中断操作码 int 调用方式进行。

5.3　调用 BIOS 中断 10H

BIOS 视频服务中断 10H 给我们提供了初始化视频模式的方法，其功能号 00H（也就是 AH 输入 00H 时）即是用于初始化视频模式，而具体要初始化的图形模式号要求输入到 AL 中（比如 13H）。中断后便进入了设置的图形模式状态。此时我们就不会受到图形驱动文件路径问题的影响了（不再需要 BGI 文件）。

我们可以通过中断调用函数 int86()（查阅所附光盘的“book\附录 F”）来触发 BIOS 视频中断 10H。

要设置图形模式 13H，中断号 intr_num 设置为 10H（BIOS 视频服务中断），inregs.h.ah 设置功能号 0H（设置视频服务模式），放入 ax 寄存器高 8 位字节中，同时将 inregs.h.al 设置为我们需要的图形模式 13H，它被放入 ax 寄存器的低 8 位字节中。此时运行 int86()函数，屏幕将被设置为 320*200（256 色）的图形模式。以下是设置显示模式的函数：

```
#define VGA256 0x13//图形模式 13h
#define TEXT_MODE 0x03//普通文本模式

void Set_Video_Mode(int mode){//设置显示模式的函数
    union REGS inregs,outregs;//定义输入和返回寄存器
    inregs.h.ah=0;//ah 存放功能号 0h 表示进入视频模式设定
    inregs.h.al=(unsigned char)mode;//al 存放要设定的视频模式
    int86(0x10,&inregs,&outregs);//通过 10h 中断设定显示模式
}
```

当参数 mode 被设置为 VGA256 的时候，函数将屏幕设置为图形模式；当 mode 被设置为 TEXT_MODE（也就是模式 03H 的 80*25 彩色文本方式），函数将屏幕设置回原先初始文本模式。

一个简单的设置视频模式的程序 setmode.c（请查阅所附光盘的“source\5”目录）。程序首先在文本模式下向屏幕输出字符，然后将视频模式设置为 13H 图形模式并在图形模式下向屏幕输出字符，最后在按键后恢复到 03H 文本模式。

5.4　用汇编设置模式

这里提出用汇编语言来设置模式的目的除了是增加一些设置显示模式的方法，还希望大家熟悉一下混合编程中汇编语言和 C 语言的接口以及掌握在 C 语言内进行行汇编。毕竟 C 语言是使用汇编写出来的，汇编的效率更高，这点对于游戏编程来说无疑是非常重要的。

虽然在之后章节设定显示模式函数中我们不会使用以下的汇编语言实现，但是，在稍候图形显示、直接写屏和内存复制的问题上我们将使用到行内汇编。所以希望大家能够通过下面的内容对汇编语言的基础有一定的了解。

5.4.1　使用汇编文件

如果你熟悉 C 语言调用汇编语言的方法，那么也可以通过效率更高的汇编语言来设置

显示模式。这里给出设置图形模式的汇编程序 SetMode.asm：

```
        .MODEL SMALL,C
        .CODE
        PUBLIC SetMode
_SetMode PROC FAR C VMode:WORD
        mov AH,0
        mov AL,BYTE PTR VMode
        int 10h
        ret
_SetMode ENDP
        END
```

其中函数名称是 SetMode，参数也是 VMode。SetVidoMode.asm 在 C 语言程序中只用如下调用即可 SetVMode.c：

```
extern SetMode(int VMode);//声明外来函数
#define VGA256 0x13
void main(void) {
    SetMode(VGA256);//调用汇编语言中的函数
}
```

然后在 DOS 提示符下键入以下命令：

```
TCC -ms SetVMode SetMode.asm
```

对 SetVMode.c 进行编译，对 SetMode.asm 进行汇编和两个文件的连接。也可以使用以下三条命令分别进行编译、汇编和连接：

```
TCC -ms -c SetVMode
TASM /Ml SetMode
TLINK c0s SetVMode SetMode, SetVMode, ,cs
```

由于 SetMode.asm 中规定了 SMALL 模式，所以它只能和按照 SMALL 模式编译的 C 模块连接。

5.4.2 行内汇编

行内汇编和使用汇编文件是不一样的。前者只是在 C 语言环境下，在一条 C 语句中调用汇编语言。

行内汇编的格式如下：

```
asm <操作码> <操作数> <; 或者换行符>
```

以下是使用行内汇编来实现显示模式设置的函数：

```
#define VGA256 0x13
#define TEXT MODE 0x03
void Set_Video Mode_Asm(int mode) {
    asm mov AH,0//ah 寄存器存放 0h
    asm mov AL,mode//al 寄存器存放图形模式对应序号
    asm int 10h//进行 10h 中断
    asm ret
}
```

也可以写成：

```
#define VGA256 0x13
#define TEXT_MODE 0x03
void Set_Video_Mode_Asm(int mode) {
asm{
    mov AH,0
    mov AL,mode
    int 10h
    ret
} }
```

其效果和使用 int86()函数是完全一样的。

无论使用哪种方法都可以设置显示模式，关键是我们在游戏中将屏幕设置成模式 13H 后，千万不要忘记在游戏结束时最终将它恢复到默认的文本模式。

5.5　本 章 小 结

在本章之前我们依赖 C 语言图形函数库来进入图形模式和绘制各类图形，为了提高代码的效率，我们决定直接通过和硬件打交道来实现图形模式和图形绘制。所有的图形绘制都必须在图形模式下实现，所以我们必须首先了解显示适配器工作原理和学会设置图形模式。

显示适配器是计算机重要的硬件组成部分。显示模式是通过对显示适配器进行设置而实现的。显示卡通常可以在各类显示模式下工作，本书建议使用图形模式 13H 来制作我们的游戏，因为其可以实现 256 色、320*200 像素，并且寻址也非常方便（显存的起始地址是 A0000H）。我们可以通过触发 BIOS 的 10H 视频服务中断 13H 功能来进入图形模式。具体的设置代码：

```
#define VGA256 0x13//图形模式 13h
void Set_Video_Mode(int mode) {//设置显示模式的函数
    union REGS inregs,outregs;//定义输入和返回寄存器
    inregs.h.ah=0;//ah 存放功能号 0h 表示进入视频模式设定
```

```
    inregs.h.al=(unsigned char)mode;//al 存放要设定的视频模式
    int86(0x10,&inregs,&outregs);//通过 10h 中断设定显示模式
}

void main(void) {
Set_Video_Mode(VGA256);//设置图形模式 13H
…}
```

从此我们可以在 13H 图形模式下开始制作丰富多彩的图形画面、动画和游戏。这将在后面的章节进行详细介绍。

学后建议

(1)尝试在计算机进入 13H 图形模式下调用 C 标准图形函数库中的画线函数 line()(也可以是其它图形绘制函数)，看看屏幕上是否有图形出现；

(2) 考虑在图形模式 13H 下显存寻址和屏幕图形之间的关系，比如在 A0000H 显存地址写入 256 中的任何一个数字，在屏幕左上角是不是就出现了一个点。那么要在右下角画点应该修改哪个显存地址呢（你可以拿出一张纸，画一个屏幕样的矩形，标注出屏幕坐标和对应的显存地址，然后随便选一个坐标点转化成显存地址）？

第6章 二维图形

本章导读

二维图形设计是游戏画面设计的基础。二维平面由两个互相垂直的坐标轴 x 轴和 y 轴组成。平面可以通过所有这两个轴上组合成的坐标点(x,y)构成。

通过本章屏幕二维图形设计我们可以更深入体会图形模式 13H 的魅力，同时对图形显示问题有一个较为全面的认知。事实上，二维图形制作和图形文件的调用是构成游戏画面的两大主要因素。第 15 章界面技术就是在二维图形的基础上实现的。

我们可以把 320*200 图形模式下的屏幕看作一个 x 轴最大为 320，y 轴最大为 200 的坐标系。所有在屏幕上面出现的图像都是由这个坐标系中的点组合产生的。

在上一章中曾经提到的绘制屏幕图形必须掌握的两点：

（1）会向屏幕画一个确定 x、y 坐标的点；

（2）懂得组成各类图形形状的算法。

本章的主要任务就是研究这两个问题的实现方式。

本章重点

（1）图形模式 13H 下直接写屏的方法；

（2）画点、线、多边形、正方形、矩形、圆和椭圆的实现算法函数；

（3）优化算法的意义和掌握编程中常用的优化思路、办法。

6.1 基本图形

二维平面中最常见的形状就是点、线和多边形。以下将依次介绍各自最高效、最简便的实现方法。

6.1.1 直接写屏

在介绍如何实现二维基本图形之前，先来介绍一种屏幕图像输出技术——直接写屏。直接写屏技术实际上就是计算出屏幕上的点所对应的显示内存位置，然后根据常规的坐标计算将图形通过点的形式保存到对应位置的显示内存中去。

这里值得强调的是，直接写屏技术最重要的两点是：

（1）直接写显存；

（2）以写点为基础操作对象。

很多人会问，第 2 章介绍到的 C 语言画点函数 putpixel()不是已经可以将点画到屏幕上去了么？

的确，C 语言也提供了画点函数和其它图形绘制函数。但是由于 C 语言是通过调用中断函数来写点的，所以效率通常不高。而直接写屏技术跳过了许多不必要的操作直接向显存对应地址输入数据大大缩短了写点时间，同时直接写屏技术也更加易于维护和计算。

上一章提到文本模式屏幕所对应的内存位置是 B8000H，同时一个字节表示一个单位文本位置，其中前面 4 位是表示文字的 ASCII 码，后 4 位是该字的属性。而图形模式（以后书中都默认模式 13H 的 320*200 像素 256 色）屏幕对应内存位置是 A0000H，由于每个像素有 256 色的选择，所以每个像素正好对应内存中的一个字节（8 位），那么从 A0000H 开始共有 320（列）*200（行）*1（字节）的内存位置对应于屏幕，于是屏幕对应的内存结尾地址是 AF9FFH。所有的直接写屏工作都是针对从 A0000H 开始到 AF9FFH 结束的显存地址进行数据写入的。

假设要向屏幕左上角写入一个白色的点，我们只需要找到显存地址 A0000H 这个字节，然后向里面写 8 位数据就可以了。问题是，到底写入什么数据？这取决于我们要输出点的颜色。0~255 所对应的 256 个颜色中，我们选择白色点对应的序号 15 写入 A0000H 就完事了。可见直接写屏的过程非常简单，就两步：

（1）找显存地址（在 A0000H 的基础上加偏移，偏移的计算通过坐标点转化而来）；

（2）写入数据（表示颜色的一个字节）。

6.1.2　直接画点

对于屏幕上的一个点的坐标(x,y)实际上就是对应于显存起始地址 320*y+x 偏移的那个字节地址。其实在上一章就提到过屏幕点的地址计算公式就是：

字节地址=A0000H+(y*320)+x

具体直接画点的完整步骤如下：

（1）用 320 乘以 Y 坐标。

（2）将相乘结果和 X 坐标相加。

（3）将相加结果作为视频内存的偏移位移量，然后加上视频内存的初始地址 A0000000H。

（4）在这个最终地址中写入你所要的颜色对应的 0~255 之间的一个值，屏幕上就显示出该点了。

其中前面三步的地址计算公式可以合并。一个画点程序 point.c 请查阅所附光盘的“source\6”目录。

此程序在屏幕上(100,100)坐标位置写入一个白色的点，注意 WHITE 事实上由 C 语言库函数定义的，其值为 15。这个程序中实现直接写屏最关键的两个语句是：

```
unsigned char far *video_buffer=(char far *)0xA0000000L;
video_buffer[((y<<8)+(y<<6))+x]=color;
```

前一句将指针变量 video_buffer 指向显存起始地址；后一句向偏移((y<<8)+(y<<6))+x（实际上就是 y*320+x）的指针变量 video_buffer 所指向的地址写入字节数据 color，从而以一个语句行的代价完成直接写点的操作。

以下给出了直接写点的函数：

```
void Plot_Pixel_Fast(int x,int y,char color) {
   video_buffer[((y<<8)+(y<<6))+x]=color;
}
```

同样从屏幕取回一个颜色点的函数正好将赋值语句中两个变量对换一下：

```
int GetPixel(int x,int y) {
    int color;
    color=video_buffer[((y<<8)+(y<<6))+x];//点的颜色等于屏幕点对应显存位置数据
    return (color);
}
```

6.1.3　直接画线

所有的直线事实上都是由点通过某种计算规则组成的。所以我们可以通过循环变量变换、画点来实现画线。

1. 竖线

画竖线不用改变 x 坐标，只要改变 y 坐标就可以了。简单来说就是从点(x1,y1)开始不断以一个步长增加（或减小）纵坐标，然后画点，直到点(x2,y2)。具体步骤描述为：

1.取得竖线最高端点显存地址；

2.画点；

3.显存地址增加 320；

4.循环步骤 2 和 3，直到竖线最低端点结束。

以下是直接画竖线的函数：

```
void VLine(int y1,int y2,int x,unsigned int color) {
    unsigned int LineOffset,index;
    LineOffset=((y1<<8)+(y1<<6))+x;//步骤 1
    for(index=0;index<=y2-y1;index++) {//步骤 4
        video_buffer[LineOffset]=color;// //改变纵坐标，步骤 2
        LineOffset+=320;//步骤 3
    } }
```

2. 横线

画横线不用改变 y 坐标，只要改变 x 坐标就可以了。具体就是从点(x0,y0)开始不断以一个步长增加（或减小）横坐标，然后画点，直到点(x1,y1)。

由于画横线点之间的地址是连续的，我们可以通过使用 memset()函数（查阅附录 D）来 copy 相同的数值到这个连续的显存位置。

画横线的具体步骤描述为：

（1）取得横线最左端点显存地址；

（2）调用 memset()函数将颜色复制到从刚才的地址开始的 x2-x1+1 个连续地址处。

以下是画横线函数：

```
void HLine(int x1,int x2,int y,unsigned int color) {
```

```
    memset((char far *)(video_buffer+((y<<8)-(y<<6))-x1),color,x2-x1+1);
}
```

这里想提醒一下的是，这个函数并不能算是直接写屏的。原因是我们调用了 memset() 函数来帮助完成这个任务。谁知道它做了些什么浪费时间的事情呢？难道没有更快的、更好的方法了么？有，在第 6.2 节中我们将详细讲这个画横线的优化方法。

3. 任意直线

画竖线和横线都只要求改变一个方向的坐标，所以只要用一个简单步长递增循环完成；而任意直线需要同时改变 x 坐标和 y 坐标。

这里有一种非常实用和易于理解的方法——Bresenham 算法。这种算法从根本上讲是通过横线与竖线的倍数关系来实现的。

具体描述为：每次循环画点后 x、y 中位移大的坐标变化一个单位；而当循环次数是两者倍数关系时，x、y 中位移小的坐标变化一个单位，就这样画出的点的集合可以近似看作一条符合要求的直线。图 6-1 和图 6-2 是 Bresenham 算法图解。

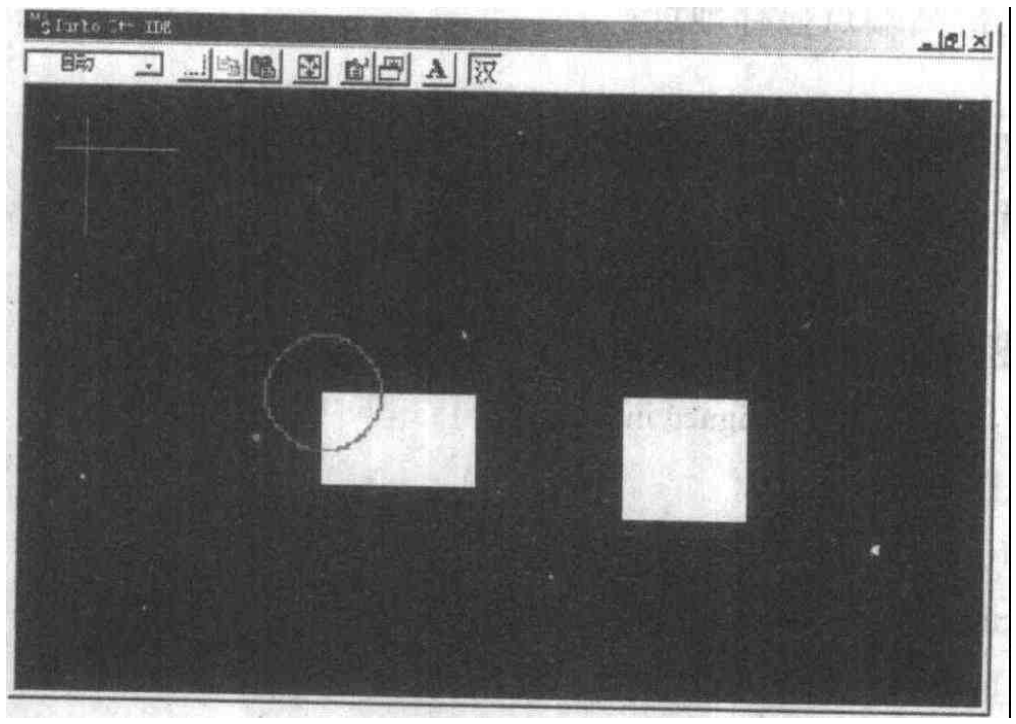


图 6-1　Bresenham 算法图解 1

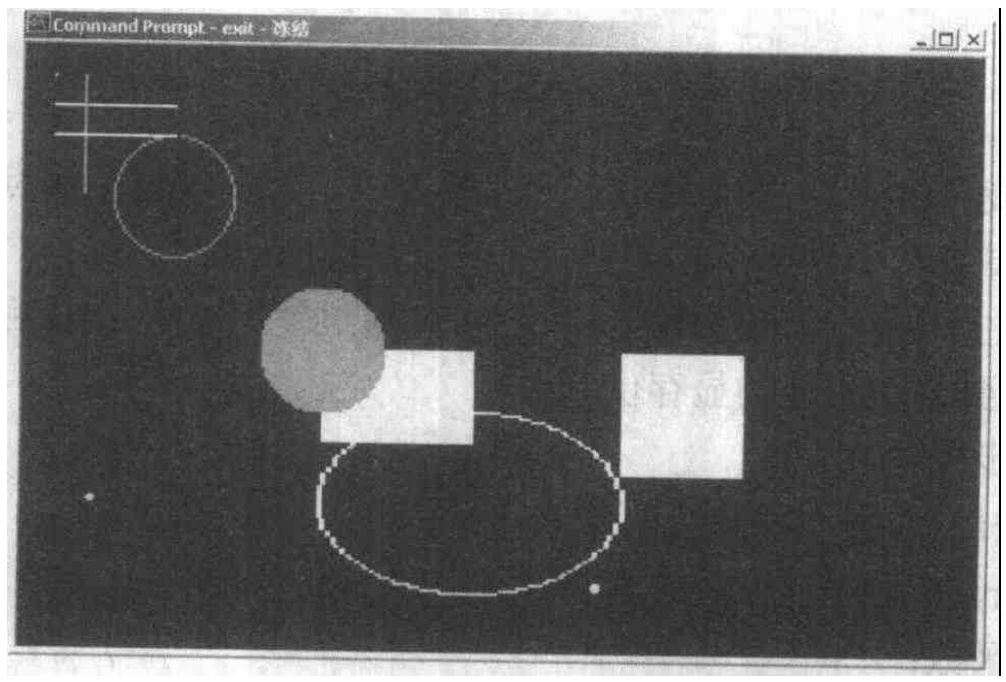


图 6-2　Bresenham 算法图解 2

下面是实现步骤：

（1）从起始点开始画点；

（2）x 位移和 y 位移中变化大的坐标加 1；

（3）循环次数加 1；

（4）判断循环次数是否是倍数关系的倍数；

（5）如果是 x 位移和 y 位移中变化小的坐标加 1；

（6）循环步骤 2~5，直到线的另一个端点结束。

举例：从点（2，2）到点（4，6），x 位移 2，y 位移 4，y 与 x 倍数关系为 4/2=2，于是先画第一点（2，2），然后 y 值加 1（倍数关系中大的坐标每次都增加 1）、画（2，3）。此时循环次数（第 2 次循环）为 y 与 x 倍数关系（2）的倍数，于是 x、y 都加 1。画（3，4），...（3，5）...，直到点（4，6）。很简单吧！这里给出实现函数 bline.h：

```
void Bline(int x0,int y0,int x1,int y1,unsigned char color) {
    int dx,dy,x_inc,y_inc,error=0,index;
    unsigned char far *vb_start=video_buffer;
    //设定起始显存位置
    vb_start=vb_start+((unsigned int y0<<6)+((unsigned int)y0<<8)+(unsigned int)x0;
    //以下获得斜线 x，y 方向上的偏移，同时明确各自偏移方向
    dx=x1-x0;
    dy=y1-y0;
    if(dx>=0) x_inc=1;
    else { x_inc=-1;
        dx=-dx; }
    if(dy>=0) y_inc=320;
    else { y_inc=-320;
        dy=-dy; }
    if(dx>dy) { //以下如果 x 长于 y 的画点循环方法
        for(index=0;index<=dx;index++) {
            *vb_start=color;//设定点的颜色
            error+=dy;
            if(error>dx) {//每循环 dx/dy 次，才满足此条件
                error-=dx;
                vb_start+=y_inc;//y 方向增加单位增量，画点
            } vb_start+=x_inc;//x 方向增加单位增量，画点
    } }
else { //以下如果 y 长于 x 的画点循环方法
    for(index=0;index<=dy;index++) {
        *vb_start=color;
        error+=dx;
```

```
        if(error>0) {
            error-=dy;
            vb_start+=x_inc;
        } vb_start+=y_inc;
    } } }
```

6.1.4 直接画多边形

多边形最关键的是边数（顶点数）和每个顶点的坐标。所以为了方便起见我们最好先建立一个多边形的结构体，然后利用这个结构体来建立我们的多边形函数。

1. 多边形结构

多边形的结构包括了顶点数、每个顶点坐标、是否填充、是否闭合、线条颜色、填充颜色等属性。结构 polygon_typ.h 定义如下：

```
typedef struct vertex_typ {//顶点结构
float x,y://顶点坐标
} vertex,*vertex_ptr;
typedef struct polygon_typ {//多边形结构
int b_color;//背景颜色
int i_color;//前景颜色
int closed;//闭合属性
int filled;//填充属性
int lx0,ly0;//起点
int num_vertices://顶点数
vertex vertices[MAX_VERTICES];//定义足够的顶点
} polygon,*polygon_ptr;
```

以上多边形的结构体除了表示边数的 num_vertices 外，还套用了一个结构体 vertex_typ。它的作用是使调用它的 polygon_typ 结构体中数组 vertices[MAX_VERTICES]的每一个成员都具有一个 x 偏移坐标和 y 偏移坐标的属性，这样所有的顶点就可以确定下来了。注意这里的偏移坐标都是针对结构中的 lx0、ly0 坐标而言的，这个坐标事实上就是多边形起始点在坐标系中的位置，也是其他偏移坐标的原点。

2. 多边形实现函数

这里的多边形函数是依靠循环调用 bline()画任意直线函数来实现的，其中每次画线的起点和终点都是相邻的结构数组点，循环后都只向下一个直到最后。这样，一圈线画下来一个多边形就成型了。

画多边形的具体步骤为：

（1）找到起始点；

（2）找到下一顶点；

（3）连接两点画线；

（4）设置下一顶点为当前点；

（5）循环步骤 2~5，直到最后一点结束。

以下为实现函数：

```
void Draw_Polygon(polygon_ptr poly) {
    int index,x0,y0;
    x0=poly->lx0;
    y0=poly->ly0;//步骤 1，找到起始点
    for(index=0;index<poly->num_vertices-1;index++) {//步骤 4.步骤 5
        Bline(x0+(int)poly->vertices[index].x,
        y0+(int)poly->vertices[index].y,
        x0+(int)poly->vertices[index+1].x,
        y0+(int)poly->vertices[index+1].y,
        poly->b_color); }//步骤 2.步骤 3，找到下一顶点，并连接两点
    if(!poly->closed)//如果不是闭合的直接返回
    return;
     Bline(x0+(int)poly->vertices[index].x,y0+(int)poly->vertices[index].y,
x0+(int)poly->vertices[0].x,y0+(int)poly->vertices[0].y,poly->b_color);
//如果是闭合的则连接最后一个顶点和第一个顶点
}
```

6.2　图形函数优化

为什么要优化图形函数呢？本来可以用 C 语言图形函数中的画线函数 line()，为什么还要自己做直接写屏的画线函数 bline()呢？就是因为效率问题。这是图形函数中比实现方法更重要的事，对吧？试想，你玩的游戏速度很慢很慢、一闪一闪的你是不是要大倒胃口了？

所以，作为最基本的图形单位的点和线更需要以最快的速度显示出来，优化势在必行！

下面一段函数是对横线函数的重新优化编制。大家有没有发现原来的程序只有一行，现在的长度足足有原先的 10 倍以上。难道这是优化么？

我们还是仔细看看它的实现思路再下结论。关键是，原先的未优化函数是通过 memset()一个字节（byte）一个字节 copy 数据来实现画横线的，而现在我们用的是一个字（word）一个字 copy。对于数据总线大于 16 位的机器而言，copy 一个字节（8 位）和 copy 一个字（16 位）的速度几乎是一样的，也就是说我们将速度提高了整整一倍多。

那么程序是如何通过字代替字节来 copy 数据的呢？看一下两者的区别：

```
unsigned char far *video_buffer=(char far *)0xA0000000L;
unsigned int far * video_buffer_w=(unsigned int far *)video_buffer;
```

我们不难发现上面一个指针是通过 char 定义的显存初始地址，而下面一个是通过 int 定义的。还记得谭浩强老师《C 语言教程》中在介绍变量类型的时候讲到过 char 是 8 位的、

int 是 16 位的么？于是，我们就不难理解指针 video_buffer 指向的是显存起始的字节，而 video_buffer_w 指向的是显存起始的字了。

问题是，如果直线的坐标不是正好能够整除 2 个字节怎么办？没关系，我们只要在对两个坐标进行判断后将最前面和最后面那两个无法整除的字节拿出来单独处理就可以了。如图 6-3 所示。

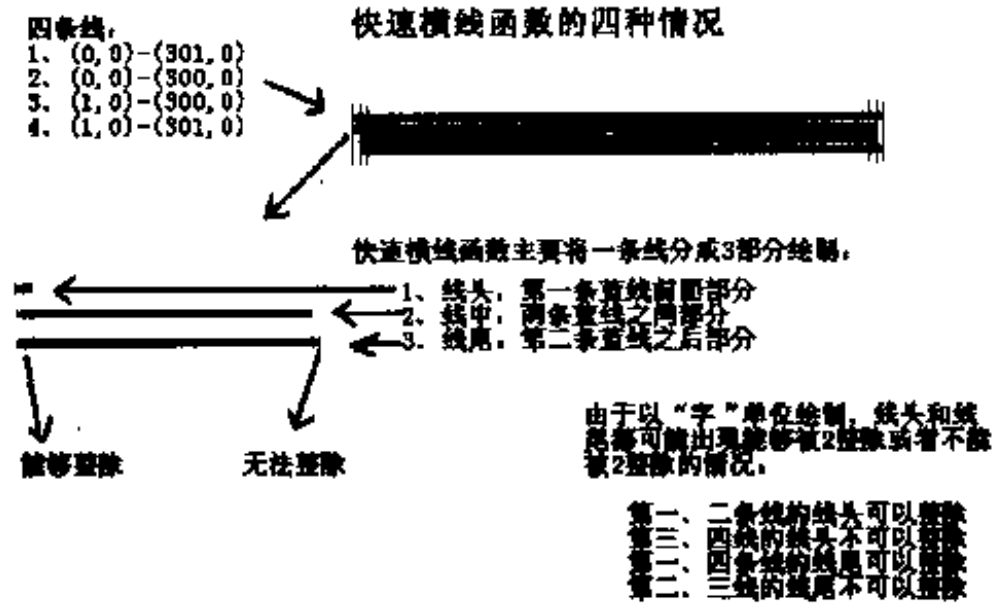


图 6-3　画线函数优化处理示意

事实上，当我们将一条横线分为三部分处理的时候，中间那部分就可以通过高速的字复制完成颜色填充。而线头和线尾部分各自存在可以整除和不可以整除的情况要处理，表 6-1 是存在情况和处理办法。

表 6-1　线头、线尾处理

情况	处理办法
线头整除	将颜色复制给表示线头的一个字变量的高位和低位
线头不整除	将颜色复制给表示线头的一个字变量的高位，低位清 0
线尾整除	将颜色复制给表示线头的一个字变量的低位，高位清 0
线尾不整除	将颜色复制给表示线尾的一个字变量的高位和低位

以下是横线优化算法：

```
void HLine_Fast(int x1,int x2,int y,unsigned int color) {
    unsigned int first_word,middle_word,last_word,line_offset,index;
    unsigned int far * video_buffer_w=(unsigned int far *)video_buffer;//字变量
    if((x1&0x0001)) first_word=(color<<8);//第一点为奇数，不整除，只填写后一点
    else first_word=((color<<8)|color);//第一点为偶数，整除，两点都填写
    if((x2&0x0001)) last_word=((color<<8)|color); //最后一点为奇数，不整除，两点都填写
    else last_word=color; //最后一点为偶数，整除，只填写前一点
    //处理直线最前面和最后面那两个无法整除的字节
    line_offset=((y<<7)+(y<<5));
    middle_word=((color<<8)|color);
    //以下以字为单位 copy 数据到显存
```

```
    video_buffer_w[line_offset+(x1>>1)]=first_word;      //画线首
    for(index=(x1>>1)+1;index<(x2>>1);index++)//以字为单位循环画线中部
        video_buffer_w[line_offset+index]=middle_word;
    video_buffer_w[line_offset+(x2>>1)]=last_word;//画线尾
}
```

6.3 更 多 图 形

在游戏制作中除了用到最多的基本图形点、线以外，我们还会使用到矩形、正方形和圆形以及它们的填充形式来作为例如窗体之类图像绘制的基础图形。以下给出这些图形的制作函数。

1. 矩形

矩形的制作是建立在画线函数基础上的。事实上我们只需要使用两个画竖线的函数和两个画横线的函数就能组成一个矩形了。以下是具体的函数：

```
void Rectangle(int x1,int y1,int x2,int y2,int color) {
    H_Line(x1,x2,y1,color);//横线函数
    H_Line(x1,x2,y2,color);
    V_Line(y1,y2,x1,color);//竖线函数
    V_Line(y1,y2,x2,color);
}
```

对于绘制填充的矩形，我们并不需要去先画一个矩形然后再考虑里面的填充问题。完全可以采用循环绘制偏移横线来完成（当然也可以是竖线），以下给出函数：

```
void Fill_Rectangle(int x1,int y1,int x2,int y2,int color) {
    int i;
    for(i=y1;i<=y2;i++) {
        H_Line(x1,x2,i,color);//横线函数
    } }
```

2. 正方形

正方形以及正方形填充问题和矩形的实现方法完全相同。以下分别给出正方形绘制函数和正方形填充函数：

```
void Square(int x,int y,int side,int color) {
    H_Line(x,x+side,y,color);
    H_Line(x,x+side,y+side,color);
    V_Line(y,y+side,x,color);
    V_Line(y,y+side,x+side,color); }
void Fill_Square(int x,int y,int side,int color) {
```

```
int i;
for(i=y;i<=y+side;i++) {
    H_Line(x,x+side,i,color);
} }
```

3. 圆

圆的绘制有一些难度，原因在于我们必须知道圆上每点的坐标变换规则。圆上每个点的具体计算主要依靠：

（1）0~360 度中每个角度的 cos 值和 sin 值；

（2）圆上点计算公式。

由于第二条是建立在第一条的基础之上的，为了减少在绘制图形中的计算工作量，建立一个角度的 cos 值和 sin 值的速查表，也就是在程序开始的时候就建立两个对应数组，以下给出建立速查表的函数：

```
void Create_Tables(void) {
int index;
for(index=0;index<=360;index++) {
cos_look[index]=(float)cos((double)(index*3.14159/180));//360 个角度 cos 数组
sin_look[index]=(float)sin((double)(index*3.14159/180));//360 个角度 sin 数组
} }
```

这个函数的主要功能是将角度转化为弧度，然后再取得 cos 值和 sin 值的数组序列。具体转换关系是：弧度=角度* π /180

在这个速查表的基础上给出圆计算公式：

```
x1=x0*cos_look[index]-y0*sin_look[index];//确定圆上 45 度点和角度时相对 x 坐标
y1=x0*sin_look[index]+y0*cos_look[index];//确定圆上 45 度点和角度时相对 y 坐标
```

其中 x0、y0 是圆内角度为 45 度时圆上点的相对坐标，x1、y1 分别是当角度为 index 度时圆上的相对点坐标。以下给出画点的具体函数：

```
void Circle(int x,int y,int r,int color) {
    int x0,y0,x1,y1,index;
    x0=y0=r/sqrt(2);//45 度时候圆上的点
    for(index=0;index<=360;index++) {//画 360 个角度时候的点
        x1=x0*cos_look[index]-y0*sin_look[index];//求出当前角度圆上相对点 x 坐标
        y1=x0*sin_look[index]+y0*cos_look[index]; //求出当前角度圆上相对点 y 坐标
        Plot_Pixel_Fast(x+x1,y+y1,color);//调用画点函数
    } }
```

请注意，此函数必须在 Create_Tables()函数被调用之后才能够使用，否则无法取出角度对应的 cos 和 sin 数组值。

以下是绘制边较粗的圆函数：

```
void BCircle(int x,int y,int r,int color) {
    int x0,y0,x1,y1,index,
    x0=y0=r/sqrt(2);
    for(index-0;index<=360;index++) {
        x1=x0*cos_look[index]-y0*sin_look[index];
        y1=x0*sin_look[index]+y0*cos look[index];
        Plot_Pixel Fast(x+x1,y+y1,color);
        Plot_Pixel_Fast(x+x1-1,y+y1,color);
        Plot_Pixel_Fast(x+x1,y+y1-1,color);
        Plot_Pixel_Fast(x+x1+1,y+y1,color);
        Plot_Pixel_Fast(x+x1,y+y1+1,color);
        Plot_Pixel_Fast(x+x1-1,y+y1-1,color);
        Plot_Pixel_Fast(x+x1-1,y+y1-1,color);
        Plot_Pixel_Fast(x+x1+1,y+y1-1,color);
        Plot_Pixel_Fast(x+x1+1,y+y1+1,color);
        Plot_Pixel_Fast(x+x1-1,y+y1+1,color);
    } }
```

圆的填充完全可以采用绘制从 0 到圆边长个圆来实现。以下给出具体函数：

```
void Fill_Circle(int x,int y,int r,int color) {
    int index;
    for(index=r;index>0;index--) {
        BCircle(x,y,index,color);
    } }
```

当然也可以不使用查表的形式来绘制圆，我们可以采用类似于绘制任意直线的思路来完成圆的制作，只是直线的斜率在每次画点后都发生了变化，最终画出 4 个 1/4 圆（不进行详细讲解，请自行研究）。

```
void B_Circle(int x_center,int y_center,int radius,int color) {
  int x;
  int y,delta;
  y=radius;
  delta=3-(radius<<1);
  for (x=0;x<=y;) {//横向和纵向分割后的四部分同时开始绘制
    Plot_Pixel_Fast(x+x_center,y+y_center,color);
    Plot_Pixel_Fast(x+x_center,-y+y_center,color);
    Plot_Pixel_Fast(-x+x_center,-y+y_center,color);
    Plot_Pixel_Fast(-x+x_center,y+y_center,color);
    Plot_Pixel_Fast(y+x_center,x+y_center,color);
```

```
    Plot_Pixel_Fast(y-x_center,-x-y_center,color);
    Plot_Pixel_Fast(-y-x_center,-x-y_center,color);
    Plot_Pixel_Fast(-y-x_center,x+y_center,color);
    if (delta<0) delta+=(x<<2)+6;
    else { delta+=((x-y)<<2)+10; y--;}
    x++;
  } }
```

4. 椭圆

```
void Plot_Circle(int x,int y,int x_center,int y_center,int color) {
  double startx,endx,x1,starty,endy,y1;
  starty=y*asp_ratio; endy=(y+1)*asp_ratio;
  startx=x*asp_ratio; endx=(x-1)*asp_ratio;
  for (x1=startx;x1<endx;+-x1) {
    Plot_Pixel_Fast(x1+x_center,y+y_center,color);
    Plot_Pixel_Fast(x1+x_center,-y-y_center,color);
    Plot_Pixel_Fast(-x1+x_center,-y+y_center,color);
    Plot_Pixel_Fast(-x1+x_center,y+y_center,color); }
  for (y1=starty;y1<endy;++y1) {
    Plot_Pixel_Fast(y1+x_center,x+y_center,color);
    Plot_Pixel_Fast(y1+x_center,-x+y_center,color);
    Plot_Pixel_Fast(-y1+x_center,-x+y_center,color);
    Plot_Pixel_Fast(-y1+x_center,x+y_center,color);
  } }

void Ellipse(int x_center,int y_center,int x_radius,int y_radius,int color) {
  register int x,y,delta;
  asp_ratio=y_radius;
  asp_ratio/=x_radius;
  y=x_radius;
  delta=3-(x_radius<<1);
  for (x=0;x<=y;) {
    Plot_Circle(x,y,x_center,y_center,color);
    if (delta<0) delta+=(x<<2)+6;
    else { delta+=((x-y)<<2)+10; y--;}
    x++;
  } }
```

一个二维基本图形的绘制例程 tuxing.c 请查阅所附光盘的“source\6”目录。

最后，图 6-4 是二维图形过程绘制效果。

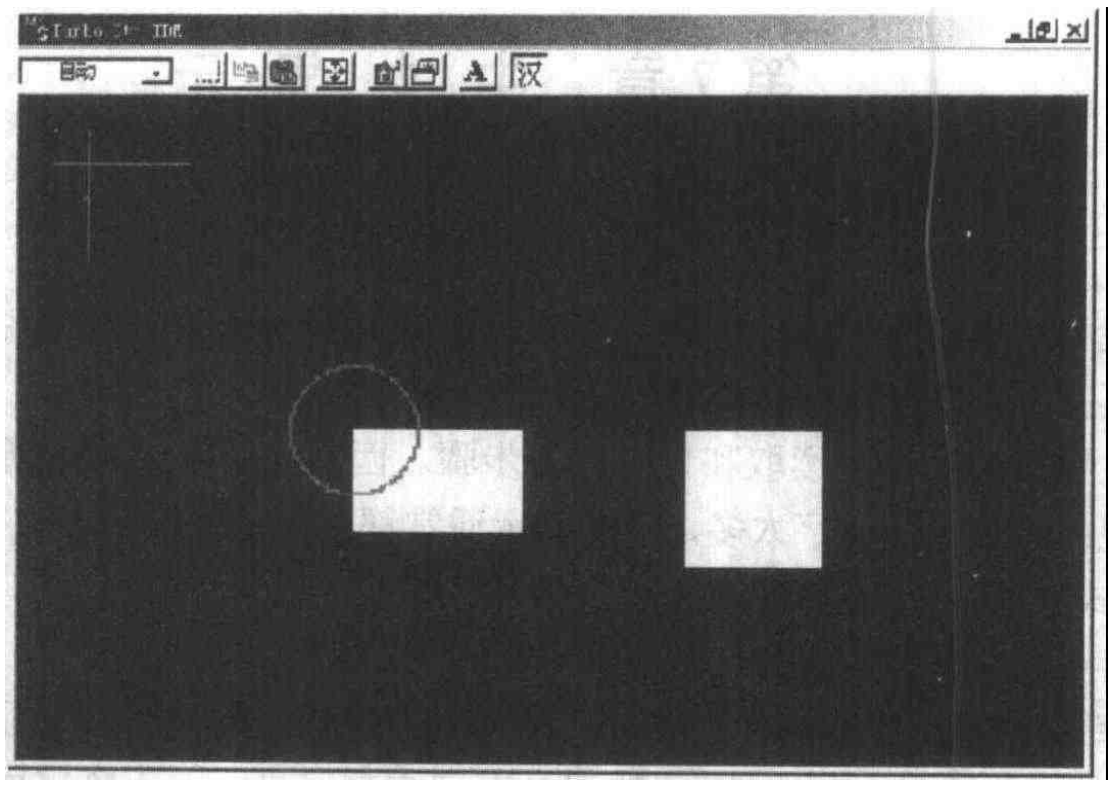


图 6-4　二维图形绘制

6.4　本 章 小 结

在进入 13H 图形模式后，我们可以通过直接写屏来绘制点。所谓的直接写屏就是向显存具体地址写入数据，从而直接在屏幕上显示出写入数据对应的颜色点。直接写屏中写具体坐标点的方法是：

（1）找显存地址（在 A0000H 的基础上加偏移，偏移的计算通过坐标点转化而来）；

（2）写入数据（表示颜色的一个字节）。

直接画点的显存地址计算公式：

```
字节地址=A0000H+(y*320)+x
```

直接画点的程序实现办法：

```
unsigned char far *video_buffer=(char far *)0xA0000000L;//显存初始地址
video_buffer[((y<<8)+(y<<6))+x]=color;//以 color 的颜色画坐标点（x,y）
```

画各类直线、多边形、正方形、矩形、圆和椭圆等二维图形都是在画点的基础上加入对应的计算规则实现的。也就是说在循环画点的过程中，每次循环都按照一定的变换规则计算出下一个点。

学后建议

（1）考虑除了横线以外的其他图形是否可以进行优化；

（2）研究二维图形的裁减、旋转、镜像和缩放等算法函数；

（3）研究封闭图形填充算法函数的实现；

（4）尝试制作更多的二维图形函数，制作一些与第 2 章 C 语言标准图形函数库中功能类似的绘图函数。

第7章　中 文 显 示

本章导读

文字在游戏中的作用是非常重要的。对于英语不好的游戏爱好者来说最害怕的就是看到游戏中全是英文。游戏文字的本地化是必须的。

文字显示主要包括文字的读取和显示两个步骤。西文显示完全可以通过调用 ROM 内存字符集来实现；但由于中文字太多，中文显示通常都要用到字库。中文字库有两大类型：

（1）点阵式字库；

（2）矢量字库。

点阵式字库通过将中文字看成由一个一个点组成的二维阵列来实现显示；矢量字库则通过对文字每个笔划的起始点和结束点的记录来完成文字显示。点阵字库和矢量字库的文件大小通常是矢量字库会小一些，尤其是在点阵较大的时候点阵字库文件的体积将远远大于大小固定的矢量字库文件。此外，点阵字库如果要放大文字将会出现明显的不平滑现象，解决的办法只有使用点阵更高的字库；而矢量字库无论字的大小都可以保证字体圆滑。

由于点阵式字库技术是中文字库主流技术，本章将着重介绍点阵式字库的显示办法，如果对矢量字库显示有兴趣的朋友可以参考汪中夏老师和苏玲老师编著的《汉字直接写屏技术》（河北科学技术出版社）一书。

本章重点

（1）点阵字库的显示原理；

（2）计算机西文 ROM 字符集显示原理和程序调用办法；

（3）中文点阵字库文件存放规则和程序显示办法；

（4）小字库和无字库技术的原理和实现办法。

7.1　文字显示原理

图形模式下西文显示的方法并不很难。比如当我们打入大写字母 X 的时候，接收函数会自动将它转化为 ASCII 码，然后对 ASCII 码进行一个简单的公式计算找到它在内存中的地址。地址找到后再将由这个地址开始的 n 个对应 X 点阵形状的字节内容 COPY 到视频缓冲中去，我们就可以看到 X 这个字母在屏幕中显示出来了。图 7-1 是一个简单的过程图。

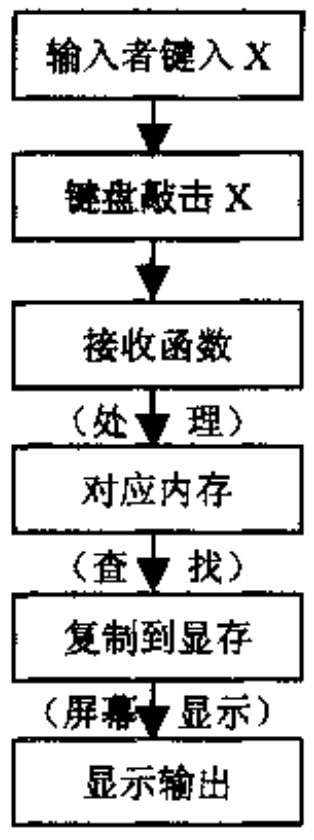


图 7-1　文字显示示意

由于英文字母加上其他一些可显示的符号一共也只有一百多个，于是我们在启动电脑的时候它们的字库就被输入到内存中去了。而我们成千上万的中文字如何显示呢？

中文显示无论是在中文环境下还是西文环境下都离不开中文字库。两者不同的是中文环境下字库事先被放到内存中去了，我们只要打入需要的中文就自动连接到内存的相应位置将它显示出来；而西文环境

下我们需要通过打开字库文件来找到我们要的汉字或者将中文字库放到足够大的内存中去。

7.2　西 文 显 示

7.2.1　使用 ROM 字符集

在说中文显示之前，我们先介绍一下西文的调用方法，西文显示默认调用 ROM 中 8*8 字符集。此外我们还可以通过 ROM 中的 8*14 和 8*16 字符集、UCDOS 提供的 ASC16 西文字库文件进行西文显示。这里主要介绍 ROM8*8 字符集西文显示。

在文本模式下（80*25）西文显示其实就是将从键盘得到的 ASCII 码放入当前光标位置所对应的显存地址（起始地址 B8000H）的第一个字节（80*25 文本模式下每个屏幕位置由两个显存地址字节对应，第一个为字符 ASCII 码，第二个为字符属性），然后计算机会自动根据显存该字节内的 ASCII 码找到 ROM 对应地址的字符集，将其显示到屏幕上。图 7-2 是 ROM8*8 字符集的描述图。



图 7-2　ROM8*8 字符集

进入图形模式，一切操作就要依靠自己了。因为图形模式下 8*8 的西文字符集的每一个位（点阵中的一个点）对应一个显存字节（一个屏幕点），而不是文本模式下的一个西文字符（ROM 中 8 个字节）对应一个显存字节（一个屏幕位置）。

具体的图形模式下西文读取步骤为：

（1）找到字符对应于 ROM8*8 字符集内的位置；

（2）找到对应于屏幕点(x,y)的显存地址；

（3）将 8 个点阵字节的每一个位读到显存地址对应的 64 个字节。

其中 ROM8*8 字符集的首地址为 FFA6EH，以下是对指向它的指针描述：

```
unsigned char far *rom_char_set=(char far *)0xF000FA6EL;
```

字符对应于 ROM8*8 集内具体地址的偏移计算公式：

```
work_char=rom_char_set+c*8;
```

其中 c 为字符的 ASCII 码，c*8 表示字符在所有字符集中的偏移地址，work_char 为字

符的具体地址。找到这个具体地址之后，我们只需要从它开始读取 8 个字节（64 个位），然后将每个位按照 0 不显示、1 显示的规则放到屏幕对应的 8*8 点阵区域就可以了。

1. 西文字符显示

显示字符函数首先要从 ROM8*8 字符集中取得当前西文字符的点阵地址，然后根据对取得的每个字节（一共 8 个字节）的每个位进行判断来决定是否向屏幕对应点的显存地址写入颜色。以下是具体显示一个西文字符的函数：

```
#define CHAR_HEIGHT 8//点阵有 8 列
#define CHAR_WIDTH 8//点阵有 8 行

void Blit_Char(int xc, int yc, char c, int color, int trans_flag) {
int offset, x, y;
char far *work_char;
unsigned char bit_mask=0x80;
work_char=rom_char_set+c*CHAR_HEIGHT;//取字符所对应字符集中的地址
offset=(yc<<8)+(yc<<6)+xc;//屏幕偏移
for(y=0;y<CHAR_HEIGHT;y++) {
bit_mask=0x80;
for(x=0;x<CHAR_WIDTH;x++) {
if((*work_char&bit_mask))//依次取出字符集地址每个字节的每个位
video_buffer[offset+x]=color;//如果位为 1 则在屏幕上显示点
else if(!trans_flag)
video_buffer[offset+x]=0;
bit_mask=(bit_mask>>1);//移位
}
offset+=SCREEN_WIDTH;//屏幕换行
work_char++;//字符集地址加 1
} }
```

2. 西文字符串显示

通过循环调用西文字符显示函数再加上一定的坐标偏移就可以显示字符串了。其中每个字的偏移就是每个西文字符的宽度 8。以下是显示一个西文字符串的函数：

```
void Blit_String(int x, int y, int color, char *string, int trans_flag) {
int index;
for(index=0;string[index]!=0;index++) {
Blit_Char(x+(index<<3), y, string[index], color, trans_flag);//调用字符显示函数
} }
```

一个简单的字符串显示程序 word.c 请查阅所附光盘的“source\7”目录。

7.2.2　使用西文字库

除了调用 ROM 内固化的西文字符集，我们也可以通过调用西文字库来显示西文。从字库文件显示字符和从 ROM 字符集中显示字符的方法都是通过字符 ASCII 码乘以每个字符点阵所占字节数来取得偏移，只是一个针对文件偏移、一个针对内存偏移，而字库文件中西文形式可以更多样。最常用的西文字库是 UCDOS 提供的 ASC12、ASC16、ASC40、ASC48 等字库文件。以 ASC16 字库文件为例，其中西文是 8*16 点阵。只需将要显示的西文所对应的 ASCII 码乘以 16 便得到了其在 ASC16 文件中的点阵起始位置，之后再连续读取 16 个字节。每个字节一行，一行 8 个点(对应字节的 8 个位)，16 个字节 16 行，每点就是字节的一个位，位如果为 1 则在屏幕上画点，为 0 则跳过。

1. 读取点阵

打开 ASC16 西文字库文件，并且根据字符 ASCII 码乘以每个西文点阵的 16 个字节取得偏移，然后读出 16 个字节到 buf 数组中，最后关闭文件。以下给出具体函数：

```
void Read_Asc16(int key,unsigned char *buf) {
int handle;
long address;
handle=open("ASC16",O_RDONLY|O_BINARY);//打开西文字库文件 ASC16
address=key*16l;//确定文件内偏移位置
lseek(handle,address,SEEK_SET);//文件内偏移，找到字符本起始位置
read(handle,buf,16);//读取该字 16 个字节的点阵到数组变量
close(handle);//关闭文件
}
```

2. 显示西文

由于 ASC16 字库存放的是 8*16 的点阵，所以在完成每个字节 8 个位的读取和写入屏幕对应位之后要进行屏幕换行操作，直到 16 个字节都读取完成。以下给出具体函数：

```
void Put_Asc16(int cx,int cy,int key,int fcolor) {
int a,b;
unsigned char buf[16];
Read_Asc16(key,buf);//调用西文读取函数
for(a=0;a<16;a++)//16 行（16 个字节）
for(b=0;b<8;b++)//8 列（每个字节的 8 个位）
if((buf[a]>>7-b)&1)//如果当前位为 1 则显示该点
VideoBuffer[(((cy+a)<<8)+((cy+a)<<6))+cx+b]=fcolor;//向显存写点
}
```

此外我们还可以通过修改西文显示函数来实现指定尺寸的文字输出，以及具有粗体、斜体和彩色格式的西文输出。一个通过调用 ASC16 字库文件来显示西文的例程 ASC16.c

请查阅所附光盘的“source\7”目录。

程序运行如图 7-3 所示。

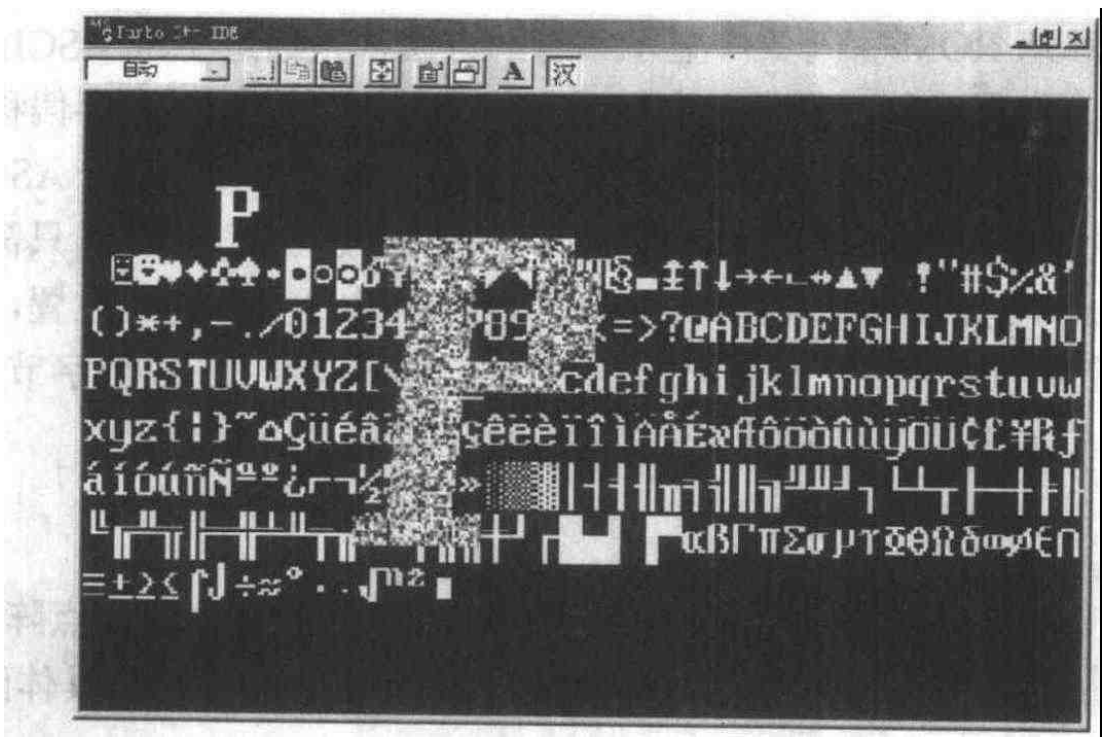


图 7-3　调用 ASC16 字库显示中文

由于此程序每次显示一个字符都要将 ASC16 字库文件打开关闭一次，这样效率很低。了解普通 ASCII 码一共有 256 个，所以其点阵所占字节一共也就 4096 个字节，查看 ASC16 字库文件后我们发现文件大小的确是 4k。于是考虑将 ASC16 字库文件一次性读入一个数组，随后的字符读取就可以对字库数组进行了。这样每次程序只用打开和关闭一次程序就可以了。

3. 读取西文字库到数组

首先我们需要在程序中申请一个 buffer 字符型数组全局变量，然后打开 ASC16 字库文件一次性将 4096 个字节全部读取到数组中。以下是具体函数：

```
char Load_Asc16() {
  int handle;
  handle=open("ASC16",O_RDONLY|O_BINARY);//打开字库文件
  read(handle,buffer,4096);//读取所有 256 个字符的点阵数据到内存
  close(handle);//关闭字库文件
  return(0);
}
```

4. 通过字库数组显示西文

我们在主程序中调用了读取西文字库到数组的函数后，就可以直接根据每个西文字符 ASCII 码乘以 16 获得其在字库数组中的起始位置，然后读取 16 个字节的点阵。以下是具体函数：

```
void Read_Asc16_Array(int key,unsigned char *buf) {
int i;
```

```
long address;
address=key*161;//计算字符在内存中偏移
for(i=0;i<16;i++)
buf[i]=buffer[address+i];//从字库数组将字符点阵读取到该字符数组中
}
```

对于字库数组西文显示问题，和之前西文字库直接调用显示的方法基本一样，只是调用读取字符点阵的函数不同而已。前者是调用 Read_Asc16()函数，这里则是调用 Read_Asc16_Array()函数。通过将西文字库读取到数组来实现西文显示的程序 ASC16arr.c 请查阅所附光盘的“source\7”目录。

7.3　中文平台下文字显示

7.3.1　汉字显示方法

中文平台下写汉字的方法很多，但基本上都是通过进入 ccbios、ucdos、spdos 和 ccdos 平台后才能进行的。

在文本模式下利用 BIOS 中断显示汉字其实就是通过 BIOS 的 10H 号中断 09 号功能显示字符，同时通过 02 号功能置光标位置；利用 C 语言库函数显示汉字就是利用窗口字符串输出函数 cprintf()和 cputs()和视频中断 10H 进行汉字显示；而直接读取视频缓冲显示汉字就是将中文字的点阵输出到文本方式下视频缓冲区 B8000H 地址中去。

在图形模式下我们依然可以使用标准输出函数，如 printf(), puts(), putchar() 函数输出中文文本到屏幕。

事实上，在中文平台下我们不需要做更多操作，一切都给我们准备好了。不过，说实话很少有游戏会在中文平台下运行的，因为并不是每台机器都是装了 UCDOS 的。

7.3.2　中文平台判别

为了提高游戏的通用性，我们可使用一个中英文平台判别程序。当判别结果是中文平台，就简单地使用标准输出函数来显示中文，否则必须使用之后介绍的西文平台下显示中文的技术。

目前最流行的中文平台就是 UCDOS，当进入 UCDOS 系统时屏幕发生了一次明显的重现现象，这事实上是因为 UCDOS 使用的是图形模式，而不是我们通常认为的文本模式。只是 UCDOS 系统耍了一个小花招，让系统以为仍然处于文本模式下。其目的就是让更多西文软件兼容于平台。

为了实现这一花招，UCDOS 将地址 00449H 处设置为文本方式（03H），同时，要使文本缓冲区和图形缓冲区同时允许使用，UCDOS 还要将 VGA 显示卡绘图控制寄存器组 GCR 中的杂项寄存器的位 2 和 3 设置为 00，而文本模式下为 11。

所以，只要 00449H 处和 GCR 杂项寄存器 2、3 位是同时为 03H 和 00 就是启动了中文平台 UCDOS，相反就是西文平台。判别例程 checkcn.c 请查阅所附光盘的“source\7”目录。

7.4　西文平台下中文调用

西文平台下汉字如何能够被显示出来呢？这里有一个简单的办法——直接调用汉字字库文件。

7.4.1　hzk16 中文字库文件

我们很容易在 UCDOS 下找到这样一个文件 hzk16，它的大小是 262k 且没有文件类型名，这是一个常用的 16*16 点阵汉字字库。我们可以通过打开这个字库文件找到需要的字并将字在图形条件下以点形式画出来。

那么，我们要的字到底在 hzk16 的什么位置呢？每个英文字母都是由一个 ASCII 码组成的，而中文字是扩展 ASCII 码组成的。也就是说一个中文字是由左半边和右半边两部分的信息组成的。那么 hzk16 中字所存放的位置和它的 ASCII 码有什么关系呢？我们可以通过以下的公式得出文字在 hzk16 中的地址：

（1）区码=文字左半边信息（扩展 ASCII 码）—A0H

（2）位码=文字右半边信息—A0H

（3）地址=[（区码—1）*94+（位码—1）]*32

其中 1、2 两步之所以将取得的扩展 ASCII 码减去 A0H 的原因是，中文字符使用到的扩展 ASCII 码是大于 A0H 的。第 3 步中（区码—1）*94 的原因是字库中每个区最多存放有 94 个中文字点阵（最后一个区不是 94 个），而乘以 32 的原因是每个中文字 16*16 点阵正好是由 32 个字节（8 位）组成。

有了具体地址又意味着什么呢？和英文点阵一样，每个中文也可以通过点组成，从这个地址开始有连续的 16 个字（32 个字节）的每个位通过 1 和 0 来表示 16*16 个点的显示和隐藏。例如

{0000H,0000H,0000H,0000H,0000H,0000H,7FFCH,7FFEH,3FFFH,0000H,0000H,0000H,0000H,0000H,0000H,0000H,}，那么它就是表示：

0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0111111111111100
0111111111111110
0011111111111111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

```
0000000000000000
0000000000000000
```

注意　0其实就是阴影（背景），而1是显示部分。

我们不难看出这实际上就是一个“一”字。

现在的任务就是将文件中的还是将字库中的内容显示到屏幕，事实上这对我们来说并不难。我们通过直接写屏的方法来实现。具体过程和英文字符显示基本相同：每次对一个字节依次从高位到低位进行屏蔽操作，如果某位为1则进行直接写屏，每次操作后显示地址增加一。两个字节后换一行显示地址就实现了。如果要字体放大，可以多增加两个循环，让每次屏蔽操作和每次字操作后都重复做放大要求的次数就可以了。

通过字库显示中文的完全过程如下：

（1）打开字库文件；

（2）找到字符在字库中的地址；

（3）通过直接写屏32*8个位写到屏幕坐标对应的16*16个字节的显存地址；

（4）循环步骤2与3，直到处理完全部字符。

这里分别给出调用hzk16库文件函数、中文字符显示函数、中文字符串函数和中文字符串渐显函数。

1. 中文字符读取

本函数用于打开hzk16字库文件，并到所要显示的字符偏移处读取32个字节到bitdata中。根据先前介绍的中文字符偏移地址计算公式，通过以下语句具体来实现偏移计算：

```
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));
```

其中ch0为中文字符左半边ASCII码，ch1为中文字符右半边ASCII码，fpos为最终偏移。以下是中文字符读取函数：

```
void GetHzBit(char ch0,char ch1,char *bitdata) {
FILE *stream;
long fpos;
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));//取中文字在文件中的偏
移
if((stream=fopen("hzk16","rb"))==NULL) {//打开hzk16字库文件
printf("Open hzk16 error!\n");
exit(0);
}
fseek(stream, fpos, SEEK_SET);//到该中文偏移位置
fread( bitdata, 32, 1, stream);//读取32个字节
fclose(stream);//关闭字库文件
}
```

2. 中文字符显示

中文显示实际上就是将刚才从中文字库读取到字符串指针地址的 32 个字节按照 16*16 点阵的规则复制到屏幕对应的显存中去。前面我们已经从一个“一”字点阵了解到需要从 32 个字节的每个位取得信息。于是在显示过程中我们通过循环位屏蔽的方法取出 32 个字节每个位的值，只要为 1 就将对应的显存地址写入文字颜色数据。以下给出中文字符显示的函数：

```
unsigned char bit[8]={128,64,32,16,8,4,2,1};
void WriteHz(char ch0,char ch1,int x,int y,int color) {
register int i,j,k;
unsigned vpos;
char bitdata[32];
GetHzBit(ch0,ch1,bitdata);//调用文字读取函数
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[2*i]&bit[j])//判断当前位是否为 1
Plot_Pixel_Fast(x+j,i+y,color);//向屏幕写入一个点
if(bitdata[2*i+1]&bit[j])
Plot_Pixel_Fast(x+8+j,i+y,color);
} }
```

3. 中文字符串显示

我们可以方便地通过循环调用中文字符显示函数并且进行一定的坐标偏移就可以实现字符串显示。由于这里使用的是 16*16 点阵字库，所以每次显示一个字符之后只需要对横坐标进行 16 个点的偏移就可以了。以下是中文字符串显示的函数：

```
void WriteHzStr(char *str,int x,int y,int color) {
int num,i,j,xx;
unsigned char s0,s1;
num=strlen(str);
xx=x;
for(i=0;i<num;i+=2){
WriteHz(str[i],str[i+1],xx,y,color);//调用字符显是函数
xx+=16;//横坐标偏移 16 个点
} }
```

4. 中文字符串渐显

中文渐显函数是在字符串显示函数的基础上改造而成的。为了提高游戏的效果，我们常常看到游戏中的文字是以一个一个间隔一定时间显示的。如同打字机效果一样（当然没有中文打字机）。我们只需要借用延迟函数就可以将文字一个一个慢慢地显示出来了。以下

是中文字符串渐显函数：

```
void Words_Step(char *str,int x,int y,int color,int speed) {
int num,i,j,xx;
unsigned char s0,s1;
num=strlen(str),
xx=x;
for(i=0;i<num;i+=2){
WriteHz(str[i],str[i-1],xx,y,color);//调用中文字符显示函数
Delay(speed);//延迟函数
xx+=16;
} }
```

值得一提的是，这里我们使用了不受计算机硬件条件影响的延迟函数 Delay()，此函数是直接通过读取存放时钟片数据的地址来获得不受计算机影响的固定时间间隔。以下给出这个延迟函数：

```
void Delay(int clicks) {
unsigned int far *clock=(unsigned int far *)0x0000046CL;
unsigned int now;
now=*clock;//读取进入延迟函数时的时间
while(abs(*clock-now)<clicks){}//延迟确定的时间
}
```

一段使用以上函数进行中文显示和渐显的程序 hzk16.c 请查阅所附光盘的“source\7”目录。

7.4.2 hzk24 中西文共显

之前我们分别提供了西文字符显示的办法和中文字符显示的办法，我们可以通过交替使用中文和西文显示函数来实现中英文共显。但是，由于这种共显无法实现一个字符串中英文共同出现的情况，所以还是有一定的缺陷。

事实上，我们可以通过在一个函数中判断当前读取的 ASCII 码是否为扩展 ASCII 码就可以分别处理了。如果是扩展 ASCII 码则调用显示中文字库，如果不是扩展 ASCII 码则调用显示西文的字库（当然也可以是 ROM 西文字符集）。而扩展 ASCII 和 ASCII 的区别就是字节最高位是否为 1，如果最高位为 1（80H）则是扩展 ASCII 码。

一个通过调用 24*24 点阵的中西文字库文件（HZK24K 和 HZK24T）来实现中西文共显的程序 hzk24.c 请查阅所附光盘的“source\7”目录。

7.5 小字库、无字库技术

无论是小字库技术还是无字库技术都是为了减小游戏文件的大小和提高游戏速度而设

计的方法。

小字库技术相当于将一个很大的中文字库文件改变成一个很小的字库文件，文件中只保留将在游戏中使用到的中文字符的点阵数据。例如一个 hzk16 字库文件的大小为 262k 左右，而提取出其中很少一部分文字的小字库文件只需要 1k~10k，这大大降低了空间的使用。

无字库技术类似于西文的字库数组，只是由于 ASCII 码一共只有 256 个，所以可以一次性全部读出到内存数组，而中文无字库技术只是将一部分被使用到的中文存放到字库内存数组。

7.5.1　小字库技术

小字库技术通过从字库文件提取需要的中文字点阵到小字库数据文件，从而使以后只需要调用小字库文件来显示中文的方法。

小字库技术的有两种：

（1）偏移小字库　通过明确知道某个中文字在小字库中的偏移位置来读取其点阵；

（2）超级小字库　较为高级，不用知道小字库中中文字偏移位置，而直接通过中文字本身的扩展 ASCII 来读取小字库中中文字点阵。

1. 偏移小字库技术

偏移小字库技术的提取可以分为单中文字符提取和中文字符串提取两种。提取过程实际上就是依次根据程序中所要求提取的中文字符串到字库文件中读取其点阵字符串到小字库文件。单中文字提取非常简单，以 16 点阵中文字为例，只需要读取 32 个字节字库点阵到小字库文件就可以了。此种小字库技术的实现步骤主要如下：

（1）决定提取哪些中文字；

（2）调用字库提取程序将中文字库文件（比如 hzk16）中的所需中文点阵保存到小字库文件中；

（3）根据具体中文在小字库中的偏移读取其点阵，并且直接显示在屏幕。

以下给出单个中文字提取函数：

```
void Hzk_File(char ch0,char ch1,char *file) {
FILE *in,*out;
long fpos;
char *bitdata;
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));//计算偏移
if((in=fopen("hzk16","rb"))==NULL){//打开字库文件
printf("Open hzk16 error!\n");
exit(0);
}
if((out=fopen(file,"wb"))==NULL){//打开小字库文件
printf("Open %s error!\n",file);
exit(0);
}
```

```
fseek(in, fpos, SEEK_SET);//找到所要读取文字偏移
fread( bitdata, 32, 1, in);//读取文字点阵
fwrite(bitdata,32,1,out);//写入小字库
fclose(in);
fclose(out);
}
```

中文字符串提取的方法实际上就是在单个字符提取的基础上加上一个循环读取要求提取的中文字符串，直到字符串的结束符（\x0）。以下给出中文字符串的提取函数：

```
void Hzks_File(char *str,char *file) {
FILE *in,*out;
char ch0,ch1;
long fpos;
char *bitdata;
int i=0,
if((in=fopen("hzk16","rb"))==NULL){
printf("Open hzk16 error!\n");
exit(0);
}
if((out=fopen(file,"wb"))==NULL){
printf("Open %s error!\n",file);
exit(0);
}
while(str[i]!='\x0') {//判断是否为结束符
ch0=str[i];//读取当前中文字左边扩展 ASCII 码
ch1=str[i+1];//读取当前中文字右边扩展 ASCII 码
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));
fseek(in, fpos, SEEK_SET);
fread( bitdata, 32, 1, in);
fwrite(bitdata,32,1,out);
i=i+2;//字符串偏移加二
}
fclose(in);
fclose(out);
}
```

一个偏移小字库提取程序 hzksave1.c 请查阅所附光盘的“source\7”目录。

经过提取后，生成了两个小字库文件，前一个 word.dat 保存了“游”字的点阵，而后一个 words.dat 存放了“游戏”两个字的点阵。

事实上，小字库提取是在游戏制作之前的一个预先工作。在完成游戏所需的中文字点阵提取之后，就可以抛弃中文字库文件，而直接使用小字库文件来读取中文字点阵来显示中文了。显示过程也可以分显示单中文字和显示中文字符串两种。

单个中文字的显示只需要明确知道你要读取的中文字在小字库文件中的位置（偏移几个中文字）就可以了。以下是单个中文字显示函数：

```
void Hzk_File_Out(char *file,int x,int y,int color,int offset) {
register int i,j,k;
unsigned vpos;
char bitdata[32];
FILE *out;

if((out=fopen(file,"rb"))==NULL){//打开偏移小字库文件
printf("Open %s error!\n",file);
exit(0);
}
fseek(out,32*offset,SEEK_SET);//根据给定的字符号码进行文件内偏移到达该字符位置
fread( bitdata, 32, 1, out);//读取 32 个点阵字符，2（列）*16（行）
fclose(out);

for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[2*i]&bit[j])//判断当前位是否为 1
   Plot_Pixel_Fast(x+j,i+y,color);//如果是 1，显示该点
if(bitdata[2*i+1]&bit[j])
   Plot_Pixel_Fast(x+8+j,i+y,color);
} }
```

中文字符串显示需要明确先前一次性提取的中文字符串数量，以及在小字库中的起始偏移位置。在显示过程中必须注意每个中文字符显示完成后必须进行点阵宽度的横坐标偏移，然后继续显示第二个中文字。以下给出中文字符串显时函数：

```
void Hzks_File_Out(char *file,int x,int y,int color,int num,int offset) {
register int i,j,k;
unsigned vpos;
int index=0;
char *bitdata;
FILE *out;
```

```
bitdata=malloc(sizeof(char)*32*num);

if((out=fopen(file,"rb"))==NULL){
printf("Open %s error!\n",file);
exit(0);
}
fseek(out,32*offset,SEEK_SET);//进行本间内偏移
fread( bitdata, 32*num, 1, out);//读取字符串中字符数量*32 个点阵字符
fclose(out);
while(index<num) {//依次显示每一个字符
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[index*32+2*i]&bit[j])
Plot_Pixel_Fast(index*16+x+j,i+y,color);
if(bitdata[index*32-2*i+1]&bit[j])
Plot_Pixel_Fast(index*16+x+8+j,i+y,color);
}
index++;
} }
```

偏移小字库显示程序 hzkload1.c 请查阅所附光盘的“source\7”目录。

2. 超级小字库技术

事实上超级小字库技术才是真正意义上的小字库技术，因为通过这种小字库技术，我们可以完全不用知道中文字在小字库文件中的位置，而如同调用字库文件一样直接写出我们需要的文字来找到它的点阵从而显示中文。

但是，超级小字库和中文字库文件的结构是不同的。这种小字库技术在提取字库文件的时候需要将被提取中文字和该中文字点阵一起放入小字库中。我们只需要将要显示中文的两个扩展 ASCII 码和小字库文件中的相比较，如果匹配就是该字的点阵。具体实现方法如下：

（1）决定提取哪些中文字；

（2）将需要的某中文字放入小字库文件；

（3）紧接着将中文字库文件（比如 hzk16）中的该中文点阵保存于其后；

（4）循环 2、3 步骤；

（5）根据中文字符到小字库文件中去匹配来读取其点阵，并且直接显示在屏幕。

具体的小字库提取函数只需要在每个中文字提取时在点阵前写入小字库两个字符的中文扩展 ASCII 码就可以了。以下给出具体的单个中文字提取和多个中文字提取函数：

```
void Hzk_File2(char ch0,char ch1,char *file) {//单中文字提取
FILE *in,*out;
```

```
long fpos;
char *bitdata;
char now[2];
now[0]=ch0;
now[1]=ch1;
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));//计算文字偏移
if((in=fopen("hzk16","rb"))==NULL){//打开中文字库文件 hzk16
printf("Open hzk16 error!\n");
exit(0);
}
if((out=fopen(file,"wb"))==NULL){//打开超级小字库文件
printf("Open %s error!\n",file);
exit(0);
}
fseek(in, fpos, SEEK_SET);//中文字库文件内偏移到该字位置
fread( bitdata, 32, 1, in);//读取 32 个点阵字符
fwrite(now,2,1,out);//写入中文字 2 个扩展 ASCII 码到超级小字库文件
fwrite(bitdata,32,1,out);写入 32 个点阵字符到超级小字库文件
fclose(in);//关闭中文字库文件
fclose(out);//关闭小字库文件
}

void Hzks_File2(char *str,char *file) {//多中文字提取
FILE *in,*out;
char ch0,ch1;
long fpos;
char *bitdata;
char now[2];
int i=0;
if((in=fopen("hzk16","rb"))==NULL){
printf("Open hzk16 error!\n");
exit(0);
}
if((out=fopen(file,"wb"))==NULL){
printf("Open %s error!\n",file);
exit(0);
}
while(str[i]!='\x0') {//只要字符串没有结束，循环提取每个中文字符
```

```
now[0]=str[i];
now[1]=str[i+1];
ch0=str[i];
ch1=str[i+1];
fpos=32L*(((unsigned char)ch0-161)*94+((unsigned char)ch1-161));

fseek(in, fpos, SEEK_SET);
fread( bitdata, 32, 1, in);
fwrite(now,2,1,out);
fwrite(bitdata,32,1,out);
i=i+2;
}
fclose(in);
fclose(out);
}
```

超级小字库具体的中文提取程序 hzksave2.c 请查阅所附光盘的“source\7”目录。

与偏移小字库技术的小字库文件比较我们发现在每个点阵前面都出现了该点阵所表示文字的 2 个扩展 ASCII 码。

由于超级小字库技术的显示函数是通过对中文字 2 个扩展 ASCII 码和小字库中所有对应位置的扩展 ASCII 码匹配来寻找到该中文字点阵的，如果小字库中没有该字的点阵则在屏幕显示没有该中文字消息。以下给出单个中文字显示函数：

```
void Hzk_File_Out2(char *file,int x,int y,int color,char *str) {
register int i,j,k;
unsigned vpos;
char bitdata[32];
FILE *out;
char str1[2];
int offset=0,flag=1;
if((out=fopen(file,"rb"))==NULL){//打开超级小字库文件
printf("Open %s error!\n",file);
exit(0);
} do {
fseek(out,34*offset,SEEK_SET);//每次偏移单位为 34，2（扩展 ASCII 码）+32（点阵）
flag=fread(str1,2,1,out);//读取扩展 ASCII 码
offset++;
}while(((str1[0]!=str[0])||(str1[1]!=str[1]))&&flag!=0);//循环判断是否有字符
if(flag!=0)//判断是否到文件尾部
```

```
    fread( bitdata, 32, 1, out); //如果没有到文件尾部，说明有，读取32个字符点阵
else printf("this chinese word can not be found");//如果到文件尾部，则说明无该字
fclose(out);//关闭字库文件
//以下是文字显示部分
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[2*i]&bit[j])
   Plot_Pixel_Fast(x+j,i+y,color);
if(bitdata[2*i+1]&bit[j])
   Plot_Pixel_Fast(x+8+j,i+y,color);
}}
```

多中文字显示函数：

```
void Hzks_File_Out2(char *file,int x,int y,int color,char *str) {
int i=0;
char now[2];
do {
   now[0]=str[i];
   now[1]=str[i+1];
   Hzk_File_Out2(file,x+i*8,y,YELLOW,now);//调用字符显示函数
   i=i+2;
}while(str[i]!='\x0');//循环显示直到中文字符串结束
}
```

超级小字库技术在程序调用中文显示的时候非常简单，我们不需要计算任何中文字在小字库中的偏移，只需要将要显示的中文字放入程序字符串变量就可以了。超级小字库显示的程序 hzkload2.c 请查阅所附光盘的“source\7”目录。

7.5.2 无字库技术

无字库技术简单地说就是通过在程序中建立一个类似字库的字库数组来实现的，它是在小字库技术基础上建立起来的。我们可以将需要的中文字点阵事先从小字库文件中放入到字库数组中，然后通过对字库数组进行偏移计算来显示我们找到我们要显示中文字。这样显示速度就大大提高了。

较小字库技术而言，无字库技术通常用于显示更少中文字的场合。同时无字库技术不需要打开任何文件，只需要对字库数组进行读取就可以了。

小字库技术的实现过程如下：

（1）字库提取；

（2）从小字库文件中取得需要的中文字数据到程序字库数组；

（3）直接通过读取字库数组来显示中文字。

由于无字库技术需要从小字库文件中抽取中文字数据到字库数组，于是小字库技术中

的提取技术也就是无字库技术的提取技术。同样无字库技术对应小字库技术也有两种：偏移无字库和超级无字库技术。

1. 偏移无字库技术

偏移无字库技术对应的是偏移小字库技术，中文字的显示完全依赖于我们知道每个字在点阵数组中的位置来实现。这种无字库技术中程序不需要出现所要显示的中文字，而只需要出现它的点阵字符串就可以了。

但是在将小字库文件内数据复制到数组中的时候，因为在小字库文件中有双引号、换行号等特殊字符，所以当这些符号被一起复制到字符串数组中去的时候程序将无法识别这些东西了。解决的办法也非常简单，按照 C 语言特殊字符处理方法，我们只需要在双引号前面加上一个反斜杠、将换行符号换成“\n”就可以了。

现将其从文件中取出放入指针数组变量，这样以后连小字库文件都不需要调用了。由于在字库文件中存在一批对于 C 程序而言对应的特殊符号（例如，引号、换行），这里对特殊符号做了必要的处理（通常可以在其之前加入一个反斜杠，换行则可以用“\n”特殊符号替换）。

对于偏移无字库技术事实上是简化了偏移小字库技术中文字的读取和显示，因为点阵已经在数组中了，而不需要从偏移小字库文件中读取。以下给出偏移无字库技术中显示单中文字函数和显示中文字符串的函数：

```
void Hzk_Array(char *bitdata,int x,int y,int color) {
register int i,j,k;
unsigned vpos;
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[2*i]&bit[j])
Plot_Pixel_Fast(x+j,i+y,color);
if(bitdata[2*i+1]&bit[j])
Plot_Pixel_Fast(x+8+j,i+y,color);
} }
```

```
void Hzks_Array(char *bitdata,int x,int y,int color,int num) {
register int i,j,k;
unsigned vpos;
int index=0;
while(index<num) {
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[index*32+2*i]&bit[j])
Plot_Pixel_Fast(index*16+x+j,i+y,color);
```

```
if(bitdata[index*32+2*i+1]&bit[j])
Plot_Pixel_Fast(index*16+x-8+j,i+y,color);
}
index++;
}}
```

偏移无字库技术具体的显示中文例程 hzkarra1.c 请查阅所附光盘的“source\7”目录。

2. 超级无字库技术

超级无字库技术应该算是真正的无字库技术。因为使用超级无字库技术显示中文无须了解字库数组中每个字的位置，而只需要将所要显示的字放入字符串变量，通过将每个中文字和字库数组中对应位置中文字比较来取出其点阵。

以下给出超级无字库技术中显示单中文字函数：

```
void Hzk_Array2(char *str,int x,int y,int color,int num) {
register int i,j,k;
unsigned vpos;
int offset=0;

while(((bitdata[offset*34]!=str[0])||(bitdata[offset*34+1]!=str[1]))&&offset<num) {
//在超级无字库数组中循环判断中文字符是否匹配直到字库数组结束或者找到该文字
offset++;
}
if(offset!=num) {//判断是否有这个字符
//如果有，以下通过读取超级无字库数组来显示这个字符
for(i=0;i<16;i++)
for(j=0;j<8;j++){
if(bitdata[offset*34+2+2*i]&bit[j])
Plot_Pixel_Fast(x+j,i-y,color);
if(bitdata[offset*34+2+2*i+1]&bit[j])
Plot_Pixel_Fast(x+8+j,i+y,color);
}
} else//否则申明没有这个字符
   printf("this chinese word can not be found");
}
```

显示中文字符串的函数：

```
void Hzks_Array2(char *str,int x,int y,int color,int num) {
int i=0;
char now[2];
```

```
do {
    now[0]=str[i];
    now[1]=str[i+1];
    Hzk_Array2(now,x-i*8,y,YELLOW,2);//调用字符读取函数
    i=i+2;
}while(str[i]!='\x0');//直到字符串结束
}
```

具体的超级无字库技术例程 hzkarra2.c 请查阅所附光盘的“source\7”目录。

7.6　中 文 特 效

7.6.1　多字体显示

为了使中文显示更加丰富多彩，我们可以使用更多的中文字体。UCDOS 为我们提供了 24 点阵的楷体、宋体、仿宋体和黑体四种字体。它们的点阵字库分别是：hzk24k、hzk24s、hzk24f、hzk24h。于是按照自己的喜好通过调用不同字库文件来显示各种字体的中文。多字体显示的例程 hzk24s.c 是在中西文共显程序上改造而成的；请查阅所附光盘的“source\7”目录。

7.6.2　文字格式显示

在 Word 中我们知道可以对文字进行各种各样的格式处理，比如粗体、斜体、放大和彩色文字等。其实这些操作都是在读取文字点阵后进行一定规则的像素偏移或者循环来完成的。具体的实现例程 hzktype.c 请查阅所附光盘的“source\7”目录。

程序运行效果如图 7-4 所示。



图 7-4　多字体显示处理

7.7 本章小结

在图形模式下西文显示仍然可以利用计算机 ROM 内存中的 8*8 西文字符集来实现。其过程非常简单：

（1）找到当前字符在存放字符 8*8 点阵的 ROM 内存中地址偏移；

（2）连续读取 8 个字符，顺序循环读取每个位，如果碰到 1 则直接画点到屏幕显存。

由于，西文字符加上其他符号最多只有 256 个（ASCII 字符 128 个，扩展 ASCII 字符 256 个），非常容易地可以固化在内存中；而中文字符有成千上万个，所以计算机中显示中文字符必须调用中文字库文件来实现。

通过字库显示中文的完全过程如下：

（1）打开字库文件；

（2）找到字符在字库中的地址；

（3）通过直接写屏 32*8 个位写到屏幕坐标对应的 16*16 个字节的显存地址；

（4）循环 2、3 步骤，直到处理完全部字符。

为了提高中文显示的速度和降低系统开销，我们可以通过多种方法实现中文显示。主要使用的方法包括：

（1）字库读取　直接通过字符在字库文件中的地址计算将点阵读出；

（2）小字库技术　从自建的相对字库文件容量小了很多（存放的中文字符少）的字库中读取点阵；

（3）无字库技术　将游戏中需要使用的字库在程序编制时就放入字库数组变量中备读取。

学后建议

（1）研究更多中文字符的特效显示方式（如对应于 Word 下针对字体的各类特效）；

（2）了解矢量字库的存放原理，尝试制作矢量字库的中文显示函数和程序。

第8章　图 形 文 件

本章导读

是否看过“仙剑奇侠传”片头云雾缭绕的仙境；是否注意过“玛丽兄弟”所有的场景都是一系列小图片组合而成；是否玩过场面宏大且富丽堂皇的“星际争霸”。所有的这些游戏的画面都无法用我们在第 6 章学到的简单的二维图形绘制而成。只有借助于其它工具我们才有机会创造如此美丽的画面。那么我们如何来创造这些画面呢？

（1）一系列好的创意：也许你可以用纸笔打下草稿；

（2）制作图片文件：你可以选择 Photoshop（制作高画质的图片）或者画笔（普通制作足够了）；

（3）你可以用纸笔画出来，然后扫描到电脑中；

（4）你可以用数码相机或者数码摄像机直接拍摄一些真实场景，然后保存为电脑图片；

（5）用 C 语言调用这些图片文件，将其显示到屏幕或者按照游戏要求进一步运用处理。

事实上，在游戏中调用图片是一件司空见惯的事情。那么有哪些最流行的图片格式呢？

我们知道最常用的图片文件就是位图文件.bmp，它可以通过 Windows 的画笔直接生成，不过它的缺点是体积较大，这对于 DOS 游戏来说是不太提倡的。

其次，pcx 图片虽然知道的人不多，但是在 DOS 游戏领域它确实绝对的“第一人”，这是由于它简单的压缩方式和较小的体积决定的，不过 pcx 图片文件的制作通常需要通过 ACDSEE 等图片软件将 bmp 等图片格式转化而来的；

这里还值得一提的是 Windows 下的图标文件.ico，这类文件的体积小、图片大小都相对固定、内部结构和制作方式也和 bmp 差不多，非常适用于类似于“玛丽”场景，这类依赖于小图片“砌砖头”的游戏和稍候介绍的子画面动画技术中。

此外，也可以尝试对 gif 图片文件和 fil 动画文件进行一些研究，因为它们的结构相对还是比较简单的；至于 jpg 格式对于游戏编程来说似乎太复杂了。

本章将着重介绍 bmp、pcx 和 ico 文件的结构、读取和显示。

本章重点

（1）Bmp、pcx、ico 三种图形文件的具体格式，包括文件头信息和数据格式；

（2）Pcx 文件编码方式，和如何对数据进行解码；

（3）Ico 文件的掩码问题（图标可以存在透明部分），及其程序实现办法。

8.1　bmp 文件调用

所谓位图就是图形在屏幕上显示时的每一个颜色都被原封不动地保存到文件的对应位置。由于是没有压缩过的位图，通常 bmp 文件比其它格式大了许多，在游戏中不太有利于使用。但是，在 Windows 中自带了画笔软件，所以编辑制作 bmp 文件容易也非常普及。本

节主要介绍 bmp 文件的调用。

8.1.1　bmp 文件结构

bmp 文件和大多数图形文件一样，分为文件描述区（头文件信息）和图像存储区（像素数据）两部分，而头文件信息中又包含了信息区和调色板区两部分，所以 bmp 文件也可以被看作三部分。信息区又可以细分为文件信息区和图像信息区两部分。用一个形象的树形结构可以表示如图 8-1。

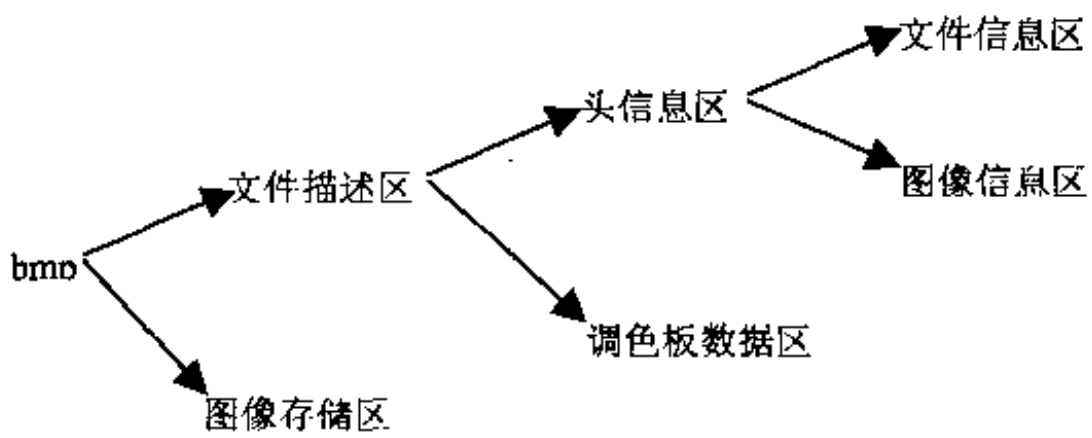


图 8-1　bmp 文件描述

这里以 256 色 320*200 的 bmp 图像为例。头文件描述区的偏移长度是 1078 个字节，也就是说图像存储区是从文件偏移 1078 后开始读取的。在头文件描述区中头信息区的偏移长度是 54 个字节，也就是说调色板数据区是从 54—1078 之间的 1024 个字节。在头信息区中文件信息区占 14 个字节而图像信息区占 40 个字节。见图 8-2。

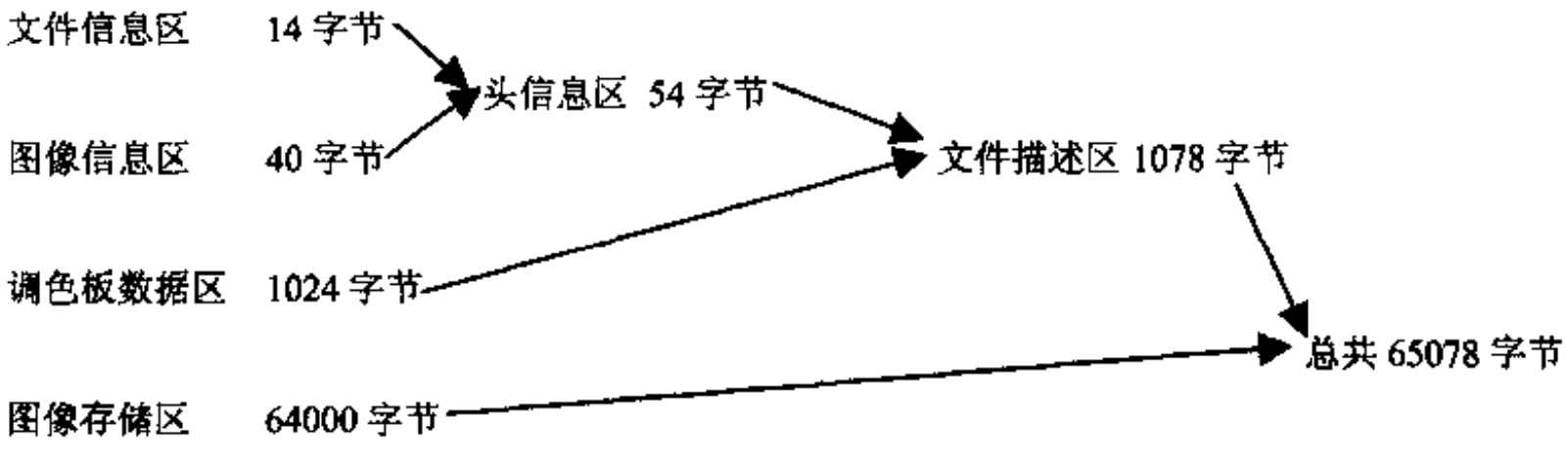


图 8-2　bmp 文件空间排列

以下给出 bmp 文件描述区的三部分的结构。

1. 文件信息区

```
typedef struct BMP_file {
    unsigned int bfType;    //文件类型，"BM",bmp 文件的标识
    unsigned long bfSize;   //bmp 文件长度
    unsigned int Reserved1;
    unsigned int reserved2;
    unsigned long bfOffset; //文件描述区长度，16 色的值为 118，256 色的为 1078
} bitmapfile;
```

我们来计算一下文件信息区结构体的长度。TC 中，int 型为 2 个字节，long 型为 4 个字节。一共是 3 个 int 型、2 个 long 型，正好是 3*2+2*4=14 个字节。

2. 图像信息区

```
typedef struct BMP_info {
    unsigned long biSize; //图形尺寸(以像素为单位)
    unsigned long biWidth; //图形宽度(以像素为单位)
    unsigned long biHeight; //图形高度(以像素为单位)
    unsigned int biPlanes;
    unsigned int biBitCount; //每个像素占二进制位数
    unsigned long biCompression; //是否是压缩格式
    unsigned long biSizeImage;
    unsigned long biXpolsPerMeter;
    unsigned long biYpelsPerMeter;
    unsigned long biClrUsed;
    unsigned long biClrImportant;
} bitmapinfo;
```

我们再来计算一下图像信息区结构体的长度。一共是 2 个 int 型、9 个 long 型，正好是 2*2+9*4=40 个字节。

3. 调色板区

```
typedef struct RGB_BMP_typ {
unsigned char blue;
unsigned char green;
unsigned char red;
unsigned char reserved;
}RGB_BMP,*RGB_BMP_ptr;
```

值得一提的是，bmp 文件调色板除了 RGB 三原色以外还有一个保留的灰度变量，于是每个颜色的调色板数据为 4 个字节（char 型是 8 位即 1 个字节），256 色就是 256 个这样的结构体变量，正好是 4*256=1024 个字节（1k）。

这里我们给 bmp 文件做一个完整的结构定义：

```
typedef struct bmp_picture_typ {
    bitmapfile file;
    bitmapinfo info;
    RGB_BMP palette[256];
    char far *buffer;
    } bmp_picture, *bmp_picture_ptr;
```

一个文件信息区结构体变量，一个图像信息区结构体变量，一个 256 色调色板结构体

变量，还有一个需要时指向存放 bmp 图形的内存的一个字符型长指针（在使用前先要申请空间，如果是直接将 bmp 文件写到屏幕上这个指针是用不到的）。

一个需要引起重视的情况是，bmp 文件图像存储区以行为单位、从下至上、从左至右的顺序对图像进行存放，同时调色板是以 B、G、R、灰度这样的倒序存放。这些将在稍后具体讲解。

8.1.2 256 色 bmp 文件显示

bmp 文件读取有两大难点：图像存储区读取和调色板区读取。这是由 bmp 文件结构在保存时不以顺序结构进行所造成的。我们首先来了解并且解决这两个问题。

1. 图像存储区的读取

由于 bmp 图像存储区数据是以行为单位、从下至上、从左至右的顺序存放的，故无法进行直接的顺序读取。

详细地说，bmp 图像存储区的数据是从 1078 偏移字节开始。文件内第一个图像点实际上是对应图像（320*200 的 bmp 图像）第 200 行的最左面第一个点，而从 1078 开始的 320 个点则是图像最下面一行对应的点，之后的 321 个点则是图像倒数第二行最左面第一个点。这样一来，bmp 文件最后一个字节对应的点就是图像最右上角（第一行最右面）的那个点了。

要将这样的文件存储结构以从左到右、从上到下的顺序读出到内存或者显存可以用以下的方法：

（1）将文件头 1078 字节开始，第一次定位到 320*199 偏移（也就是图像最后一行首）位置；

（2）读取第一行 320 字节的数据到内存；

（3）定位位置较刚才偏移位置少 320 字节（也就是到图像刚才行的上一行首）位置；

（4）读取该行 320 字节的数据到内存；

（5）循环 3、4 步骤，直到读完第一行图像数据。

以下给出实现 bmp 文件图像存储区数据读取到内存的代码段：

```
for(i=SCREEN_HEIGHT1-1;i>=0;i--) {
//一共进行屏幕高度次数的文件读取
    lseek(fp,1078+(long)(SCREEN_HEIGHT1-i-1)*SCREEN_WIDTH,0);
    //定位到文件内当前读取行首，定位倒序
    read(fp,&bmp256->buffer[i*SCREEN_WIDTH],SCREEN_WIDTH);
    //从下到上复制文件屏幕宽度内容到屏幕，屏幕倒序
}
```

2. 调色板的读取

除了图像存储区的存放规则是到序的以外，bmp 文件调色板区内容也是以 B、G、R、灰度的顺序存放的。在读取的时候需要非常仔细地读取对应的三原色，而不可以将文件中的三原色的蓝色对应给调色板结构体变量的红色。

同时由于每种三原色都只使用了 64 种（6 位）色阶，而给每种三原色存放的空间是 1 个字节(8 位，可存放 256 个色阶)。于是 bmp 文件将每种三原色的 6 位数据都放在 1 个字节（8 位）的高位，使得低 2 位为 0。在读取到调色板结构体变量的时候必须进行右移 2 位操作（也可以除以 4）。

以下给出读取文件调色板区数据的代码段：

```
for (i=0;i<256;i++) { //256 色，要读取 256 个三原色
 read(fp,&bmp256->palette[i].blue,1); //首先读取蓝色
 read(fp,&bmp256->palette[i].green,1); //然后读取绿色
 read(fp,&bmp256->palette[i].red,1); //最后读取红色
 read(fp,&bmp256->palette[i].reserved,1); //读取灰度数据
 //将在高位的 6 位数据移到低位
 bmp256->palette[i].blue=bmp256->palette[i].blue>>2;
 bmp256->palette[i].green=bmp256->palette[i].green>>2;
 bmp256->palette[i].red=bmp256->palette[i].red>>2;
}
```

以下是写入调色板函数：

```
void Set_BMP_Palette_Register(int index,RGB_BMP_ptr color) {
outp(PALETTE_MASK,0xff);
outp(PALETTE_REGISTER_WR,index);//确定要设定的调色板序号
outp(PALETTE_DATA,color->red);//设置该序号的红色
outp(PALETTE_DATA,color->green);//设置该序号的绿色
outp(PALETTE_DATA,color->blue);//设置该序号的蓝色
}
```

调用调色板的代码段：

```
for (i=0;i<256;i++)//循环设置 256 个调色板数据
    Set_BMP_Palette_Register(i,(RGB_BMP_ptr)&bmp256->palette[i]);
```

请注意，在写入调色板函数中灰度并没有被使用到。而写入计算机调色板的顺序还是 R、G、B，而不是 bmp 文件中的 B、G、R。

在解决了 bmp 文件读取的两大难点之后，我们可以开始对 bmp 文件进行完整的读取了。bmp 文件的显示过程包括以下几个步骤：

（1）申请内存空间；

（2）检查头文件信息区；

（3）读取调色板数据；

（4）读取位图到内存；

（5）显示图像；

（6）内存释放。

3. 申请内存空间

当希望将 bmp 文件通过内存显示到屏幕的时候，首先必须申请适合图像大小的内存空间。至于文件头数据和调色板数据所需要的内存我们早在定义 bmp 文件结构变量的时候已经以数组等形式申请好了。

bmp 图像大小空间在这里事实上就是 320*200 个字节，我们只要使用 malloc()内存申请函数进行申请就可以了。

以下给出申请 bmp 文件内存的函数：

```
void BMP_Init(bmp_picture_ptr image) {
//申请图像大小的内存空间
unsigned int a=(unsigned int)(SCREEN_WIDTH * SCREEN_HEIGHT + 1);//计算内存大小
if((image->buffer = (char far *)malloc(a))==NULL) {//申请画面要求大小的内存
   printf("\ncouldn't allocate screen buffer");
   exit(1);
 } }
```

这里需要提出的是一次申请 320*200（64K）的空间在实际操作中是失败的，解决它的方法有几种：

（1）直接读取文件到屏幕（这将在稍后讲解）；

（2）将图像读取到 EMS 或者 XMS 中去（这将在第 13 章内存技术中详细讲解）；

（3）每次读取少量内容到足够申请到的内存（比如 320*50），反复进行几次操作，全部读出。

4. 检查头文件信息区

检查 bmp 文件信息区的主要任务是：

（1）判断文件是否是 bmp 文件（不是则无法显示）；

（2）文件是否是压缩 bmp 格式（是压缩的则不处理）；

（3）文件是否是 256 色。

以下给出读取头文件信息到内存的代码段：

```
read(fp,&bmp256->file,sizeof(bitmapfile)); //读取文件信息区
read(fp,&bmp256->info,sizeof(bitmapinfo)); //读取图像信息区
```

以下给出检查 bmp 文件格式的函数：

```
void Check_Bmp(bmp_picture_ptr bmp_ptr) {
if(bmp_ptr->file.bfType!=0x4d42) { //检察是否是 bmp 文件
   printf("Not a BMP file!\n");
exit(1);
}
if(bmp_ptr->info.biCompression!=0) {
```

```
//检查是否是压缩格式,1 表示压缩 0 表示没有压缩
printf("Can not display a compressed bmp file!\n");
exit(1);
}
if(bmp_ptr->info.biBitCount!=8) { //检查是否是 256 色
printf("Not a index 16color bmp file!\n");
exit(1);
} }
```

这里需要介绍的是，在检查是否是 bmp 文件的时候是判断两个字节是否是 bmp 标志“BM”，“BM”16 进制表示为“424d”,由于存放在 int 型变量 ptr->info.biType 里面，高位在左所以是“4d42”。此外，再检查是否是 256 色的时候，由于 8 位正好是 256 色，所以就是判断变量 bmp_ptr->info.biBitCount 是否等于 8,如果是 16 色此变量则等于 4。

5. 显示图像

在将 bmp 调色板区写入计算机调色板和已经读取 bmp 文件图像存储区到内存的基础上，如何将内存数据显示到屏幕上呢？

其实我们通常所说的屏幕就是显存。而从内存将数据给显存只要进行一个内存复制操作。

以下给出具体的复制函数：

```
void BMP_Show_Buffer2(bmp_picture_ptr image) {
memcpy((char far *)video_buffer,(char far *)image->buffer,
(unsigned int)SCREEN_WIDTH*SCREEN_HEIGHT/2);
}
```

其中 image 是 RGB_BMP_typ 结构变量，image->buffer 当然就是指向 bmp 图像所在的内存的指针。而 video_buffer 是指向显存起始位置 A0000000H 的指针。每次 memcpy()函数是以字（2 个字节）的单位进行复制，所以要复制的字的大小为 SCREEN_WIDTH*SCREEN_HEIGHT/2。

是否还有其他更快的方法？要知道复制 64000 个点到屏幕的确不是一件很小的事情，越快越好！

更快更好的方法是有的，事实上 TC 提供给我们的库函数也都是由更低层的汇编语言写成。我们可以直接在 C 语言中使用汇编语句来实现同样的效果，应该说这样的速度会快不少。

以下给出另一种调用汇编语句来实现内存向显存复制的函数：

```
void BMP_Show_Buffer(bmp_picture_ptr image) {
char far *data;
data=image->buffer; //将 data 指向存放 bmp 内存的位置
asm push ds; //DS 寄存器压栈
```

```
asm les di,video_buffer; //目的寄存器放入指向显存起始位置的指针
asm lds si,data; //源寄存器放入指向存放 bmp 内存的指针
asm     mov cx,SCREEN_HEIGHT*SCREEN_WIDTH/2;
//CX 寄存器放入要复制的字数
asm     cld; //调用内存复制语句
asm rep movsw; //循环复制直到 CX=0
asm pop ds; //DS 寄存器出栈
}
```

对于汇编语言我们在第 5 章已经有所介绍，这里再次强调一下，如果要在 C 语言中插入汇编语句只需要在每句汇编语句前加上 asm。还有一种办法可以在一段汇编语句的前面加上 asm，并且用大括号括起来，如下：

```
    asm{
        pop ax
        pop ds
        iret
    }
```

此外，无论使用那种方法在汇编语句行最后加或者不加分号都是可以的。比如：

```
void BMP_Show_Buffer(bmp_picture_ptr image) {//将内存中的图像显示
char far *data;
data=image->buffer;

asm push ds
asm les di,video_buffer
asm lds si,data
asm     mov cx,SCREEN_HEIGHT*SCREEN_WIDTH/2
asm     cld
asm rep movsw
asm pop ds
}
```

这样编译也是可以通过的。

6. 释放内存

释放内存的函数非常简单，我们只需要调用一个 free()函数就可以了。释放 bmp 文件内存的函数如下：

```
void BMP_Delete(bmp_picture_ptr image) {
free(image->buffer);//释放对应图像申请的内存空间
}
```

以下给出 2 个具体的显示 bmp 文件的例程：

（1）使用内存调用方法显示 bmp 文件；

（2）直接从文件读取到屏幕显示 bmp 文件。

内存调用方法显示 bmp 文件 bmpmem.c 请查阅所附光盘的“source\8”目录。

值得一提的是通过内存显示 bmp 文件受到常规内存申请的限制。在本程序中我们事实上只申请了 320*40 的内存空间，而 256.bmp 这个文件的实际大小是 320*200，如此大的空间是无法一次性申请到的。

解决无法申请足够空间来显示图像文件的办法有两种：

（1）直接将文件写入显示内存；

（2）将文件读取到扩展、或者扩充内存（这将在稍后的动画和内存章节中介绍）。

直接将图像文件读取到显存的方法非常简单，它将通过内存显示 bmp 文件所需要的申请空间、读取数据到申请的空间、显示等多步任务简化成一步。事实上也就是将读取文件数据到申请内存变成到显示内存。直接从文件读取到屏幕的程序 bmp.c 请查阅所附光盘的“source\8”目录。256.bmp 图像如图 8-3 所示。

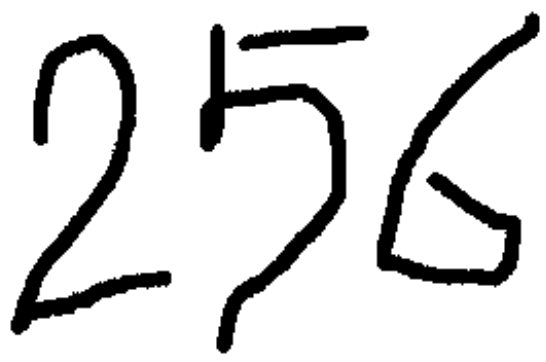


图 8-3

8.2　pcx 文件调用

8.2.1　pcx 文件结构和编码

pcx 文件是一种游戏中使用得非常流行的图形文件，它的特点是文件很小，编码简单。对于一个 320*200*256 的图形，我们无法想象将整整 64K 的内容原封不动地保存到文件中去，事实上如果这样做的话太浪费了。pcx 文件给出了很好也很容易懂的压缩方法。

首先，pcx 文件包含三部分：

（1）pcx 文件第一部分占据 128 个字节，放置各种信息；

（2）第二部分是压缩图像数据部分，任意大小；

（3）最后的 768 个字节是该文件调色板数据。

3 部分寻址都非常简单。第一部分以打开文件就可以读了；第二部分如果你对读 128 个头文件字节非常有信心就可以直接读下去或者干脆决定放弃第一部分头文件不读用 fseek()函数对 SEEK_SET(文件头)偏移 128 就可以读了；第三部分你当然也可以继续读下去，不过实际上最常用的还是 fseek()函数对 SEEK_END(文件尾)偏移 768 个字节（有没有注意到 768 字节就是 256 个三原色，每个三原色占 1 个字节）的读取。

先来看看头文件的结构体如下：

```
typedef struct pcx_header_typ {
    char manufacturer;//制造厂
    char version;//版本
    char encoding;//编码方式
```

```
char bits_per_pixel;
int x,y;
int width,height;//图像宽度和高度
int horz_res;
int vert_res;
char ega_palette[48];
char reserved;
char num_color planes;
int bytes_per_line;
int palette_type;
char padding[58];
} pcx_header, *pcx_header ptr;
```

让我们来算一算有多少字节。在 tc 中，char 型占 8 位即 1 个字节，int 型占 2 个字节。算上数组一共有 6+48+58=112 个 char 型和 8 个 int 型变量。112+8*2=128 个字节，正好是头文件的偏移数。

```
typedef struct pcx_picture_typ {
   pcx_header header;//pcx 头部分
   RGB_color palette[256];//调色板数据部分
   char far *buffer;//数据区指针
   } pcx_picture, *pcx_picture_ptr;
```

与先前 bmp 的结构相同，pcx 文件结构中也是一个头文件结构变量，所有的头文件都读到这里面去；一个 256 色调色板数组变量，pcx 文件末尾的该调色板数据读到这里来；一个字符型长指针，需要时指向存放图形的内存（在使用前先要申请空间，如果是直接将 pcx 文件写到屏幕上这个指针是用不到的）。

这里要着重介绍 pcx 文件的压缩编码技术。首先希望大家不要一听到编码或者压缩就马上皱起眉头认为一定好难好难的，其实有些编码方式非常简单也非常有趣。比如这里的 pcx 编码就是很符合我们普通的思维方式的。

先来问一些关于图形文件大小的问题。

当你刚刚用画笔画了一幅由大片蓝天和海洋组成的 320*200 图画后，你决定保存它。好吧，就用 256 色 bmp 保存（事实上也只能用 bmp 保存）。我们在上一节介绍 bmp 文件的时候说过了它的保存大小是多少？

位图嘛，320*200 个点，每个点对应 1 个字节（8 位=256 色），正好 64000 个字节。再加上头文件和调色板数据，bmp 文件保存竟然要差不多 64k。

于是要你自己想一个压缩编码方式将空间需求减下来，你会怎么想？

对了，看到蓝天还有海洋都是大片大片蓝色的，那么干脆算一下有多少连续的相同蓝色。来数数看（当然是让程序去做，你数的话会数死的），这里有 40 个连续的蓝色。原本这里要用 40 个字节存放蓝色的序号，现在我们只需要用 40 和蓝色序号 2 个字节来存放，

足足省下了 38 个字节。估计一个文件下来最多也就几 k 大小，至少是 bmp 文件的 1/10。

对于 pcx 文件在 128 个字节头文件偏移后的每个字节有这样的解压规则：

（1）如果读出的字节在 192~255 之间，就用这个数字减去 192，减出的结果就是下一个字节表示的色彩重复的次数；

（2）如果读出的字节在 0~191 之间的范围，就直接是下一个颜色。

注意　这里有一种情况是两解的，比如：颜色 45 可以表示为（45）或者（193，45）。因为 193-192=1 也就是说它后面的 45 重复的次数为 1，完全等于用一个 45 表示。这完全是因为程序编码的方法不同出现的。此外，如果我们要将 211 对应的颜色显示在屏幕上，就应该用（193，211）来表示。

下面是一个解压的例子：

203，4，34，193，213

203：203-192=11

4：11 个 4

34：1 个 34

193：193-192=1

213：1 个 213

于是，解压结果为：

4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 34, 213

知道刚才的漏洞是什么了吧？其实就是当下次看到 40 和蓝色序号的时候别人不一定知道这是表示 40 个蓝色，可能以为是 40 这个序号对应的颜色（也许是红色）和蓝色两个点。

为了避免这种歧义，pcx 压缩算法只使用了 256 中的 191 个颜色，而在 192 以后的将不会是颜色则可能有一种理解：颜色重复数量。当然这个重复数量最小就是 192，所以减去 192 才是真正的数量（40 个蓝色表示为：232，蓝色的序号 这两个字节）。

下面是解压过程（解码到内存）的 C 语言实现方法的部分代码段：

```
data = getc(fp); //读 pcx 文件中一个数据段字节
if (data>=192 && data<=255) { //判断是否在 192~255
num_bytes = data-192;
data  = getc(fp); //读出重复次数
while(num_bytes-->0) { //读出下一个字节，即颜色
 image->buffer[count++] = data; //将重复的颜色数据解压到内存
 } } else { //如果读到的字节在 0~191 之间
image->buffer[count++] = data, //直接将颜色数据给到内存
} }
```

注意　如果你准备将 pcx 图像数据解压显示到屏幕，只需要将 image->buffer 改成指向屏幕起始的指针变量 video_buffer 就可以了。

8.2.2　pcx 文件显示

已经攻克了压缩编码这个不算难的难关，现在要显示 pcx 文件就变得非常简单了。还记得上节说到的 pcx 文件的另外两个部分么？对于头文件我们可以忽略不去处理，还剩下一件事情就是读文件最后 768 个字节的调色板数据。

读 pcx 文件调色板数据比起读 bmp 数据要简单很多。原因：

（1）pcx 文件调色板数据是以 RGB 顺序存放的，而 bmp 文件是反的；

（2）pcx 文件调色板数据只有 RGB 三部分，而 bmp 文件则多了一个保留用的灰度。

读取 pcx 文件调色板数据并且写入计算机调色板的代码段如下：

```
fseek(fp,-768L,SEEK_END); //到文件末第 768 个字节
for (index=0; index<256; index++) { //读取 256 个三原色
image->palette[index].red   = (getc(fp) >> 2); //读取红色
image->palette[index].green = (getc(fp) >> 2);    //读取绿色
image->palette[index].blue  = (getc(fp) >> 2); //读取蓝色
}
if (enable_palette) { //pcx 文件调色板开关
  for (index=0; index<256; index++) {//要写 256 个颜色
    Set_Palette_Register(index,(RGB_color_ptr)&image->palette[index]);
   //写入计算机调色板
  } }
```

这里值得注意的是，如果不打开调色板开关，读入到内存的 768 个字节的调色板数据将不被调入计算机调色板，计算机在显示 pcx 图像时仍然实用原先的计算机调色板数据。实际情况是，如果你不打开调色板往往写入屏幕的 pcx 文件颜色是不对的，也就是说文件调色板和计算机调色板通常不会一样。只有一种情况：你第一次将 pcx 文件调色板调入计算机调色板，之后显示的许多 pcx 文件（是在第一个 pcx 文件绘制基础上修改的文件）不打开调色板开关也能出现正确的颜色。

在了解 pcx 文件 3 部分结构之后我们来完整地策划一下显示 pcx 文件的方法。事实上，pcx 文件的显示包括以下要处理的几个顺序部分：

（1）申请内存空间；

（2）取出压缩数据和调色板到内存；

（3）将内存中的数据显示；

（4）释放内存。

由于和前面介绍的 bmp 文件显示的具体方法相类似，这里只给出具体的函数而不再进行一一赘述了。

以下给出申请内存空间的函数：

```
void PCX_Init(pcx_picture_ptr image) {
unsigned int a=(unsigned int)(SCREEN_WIDTH * SCREEN_HEIGHT + 1);
if((image->buffer = (char far *)malloc(a))==NULL) {
```

```
  printf("\ncouldn't allocate screen buffer");
 exit(1);
 } }
```

以下给出将 pcx 文件装入内存的函数：

```
void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette) {
FILE *fp,
int num_bytes,index;
unsigned int count;
unsigned char data;
char far *temp buffer;

fp = fopen(filename,"rb");
temp_buffer = (char far *)image;

for (index=0; index<128; index+-) {
temp_buffer[index] = getc(fp);
} count=0;

while(count<=(unsigned int)SCREEN_WIDTH * SCREEN_HEIGHT) {
    data = getc(fp);
 if (data>=192 && data<=255) {//以下解压缩处理
num_bytes = data-192;
data  = getc(fp);

while(num_bytes-->0) {
 image->buffer[count++] = data;//压缩点数据存入内存
 } else {
image->buffer[count++] = data;//非压缩点数据存入内存
} }
//以下读取文件尾的调色板数据
fseek(fp,-768L,SEEK_END);

for (index=0; index<256; index++) {
image->palette[index].red   = (getc(fp) >> 2);
image->palette[index].green = (getc(fp) >> 2);
image->palette[index].blue  = (getc(fp) >> 2);
} fclose(fp);

if (enable_palette) { //以下设置计算机调色板
   for (index=0; index<256; index++) {
   Set_Palette_Register(index,(RGB_color_ptr)&image->palette[index]);
```

```
}}}
```

以下给出将内存中图像显示到屏幕的函数：

```
void PCX_Show_Buffer(pcx_picture_ptr image)
{
char far *data;
data=image->buffer;

asm push ds;
asm les di,video_buffer;
asm lds si,data;
asm     mov cx,SCREEN_HEIGHT*SCREEN WIDTH/2;
asm     cld;
asm rep movsw;
asm pop ds;
}
```

以下给出释放内存空间的函数：

```
void PCX_Delete(pcx_picture ptr image)
{
free(image->buffer);
}
```

一个完整的内存调用 pcx 文件显示的程序 pcxmem.c 请查阅所附光盘的“source\8”目录。

和 bmp 文件的显示一样，pcx 文件的显示通常是先将 pcx 读入内存然后复制到屏幕所在的显存，但在实际操作中要申请如此大（320*200，64k 左右）的内存空间仍然是非常困难的。此程序中考虑到无法申请到 320*200 的内存空间，于是决定申请 320*40 的空间。如果要显示 320*200 的 pcx 文件则可以将 pcx 文件直接复制到显存中对应位置。

将 pcx 直接显示到屏幕上，要对刚才的功能函数做以下几点修改：

（1）不需要内存申请函数和内存释放函数；

（2）将原先 pcx 文件调入申请内存的函数改为直接调入显存，不需要原先的图像显示函数了。

直接显示一张 320*200 的 256 色 pcx 文件的程序 pcx.c 请查阅所附光盘的“source\8”目录。

8.2.3 播放 pcx 文件

如果我们要做一个相对内容丰富的动画片头，可以考虑用 pcx 文件连续显示的方法实现。具体的制作方法很简单：

（1）用 Windows 画笔软件画一个基本片头图画；

（2）然后使用复制、绘画等方法对其进行渐变操作，每一个变化保存到一个图像文件中；

（3）使用 acdsee 图像软件将 bmp 图像转换成 pcx 文件；

（4）将 pcx 图像文件序列名加入显示程序中，同时将显示程序改成一个循环显示语句，每次延迟足够时间。

一个简单的 256 色 pcx 文件播放程序 pcxs.c 请查阅所附光盘的“source\8”目录。

8.3　ico 文件显示

8.3.1　ico 文件结构

由于 ico 文件体积小（16 色 32*32 点阵的通常小于 1k）、标志性强等优点在 Windows 中程序中得到广泛使用。如果我们在游戏中配合 ico 文件调用将使游戏更加直观、明白。

这里介绍如何调用标准的 ico 文件（以最常用 16 色 32*32 点阵的 ico 图标为例，非画笔制作的 ico 文件）。ico 文件也分为文件结构区、调色板数据区和文件数据区。其中 ico 文件的数据区和 bmp 有些类似，只是增加了一个掩码区。同时 ico 文件长度通常是 766 个字节。ico 文件内部结构如图 8-4 所示。

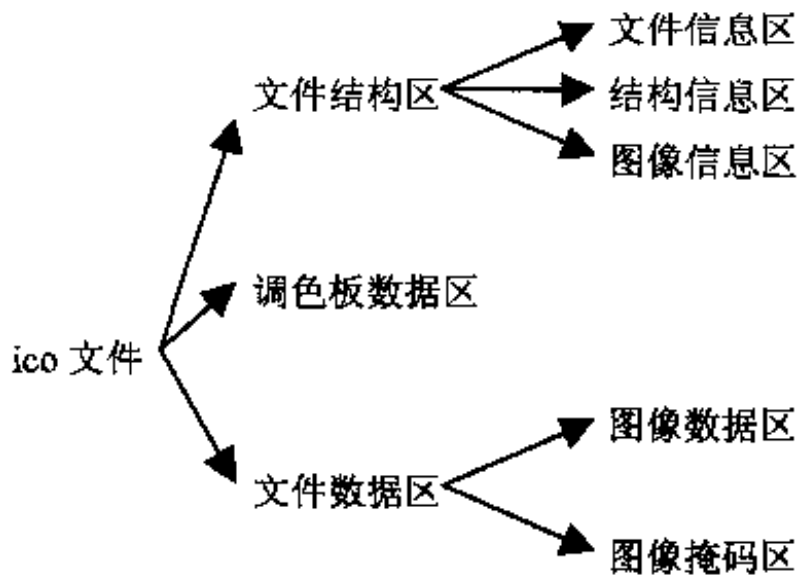


图 8-4　ico 文件结构

ico 文件各部分字节占用如图 8-5 所示。

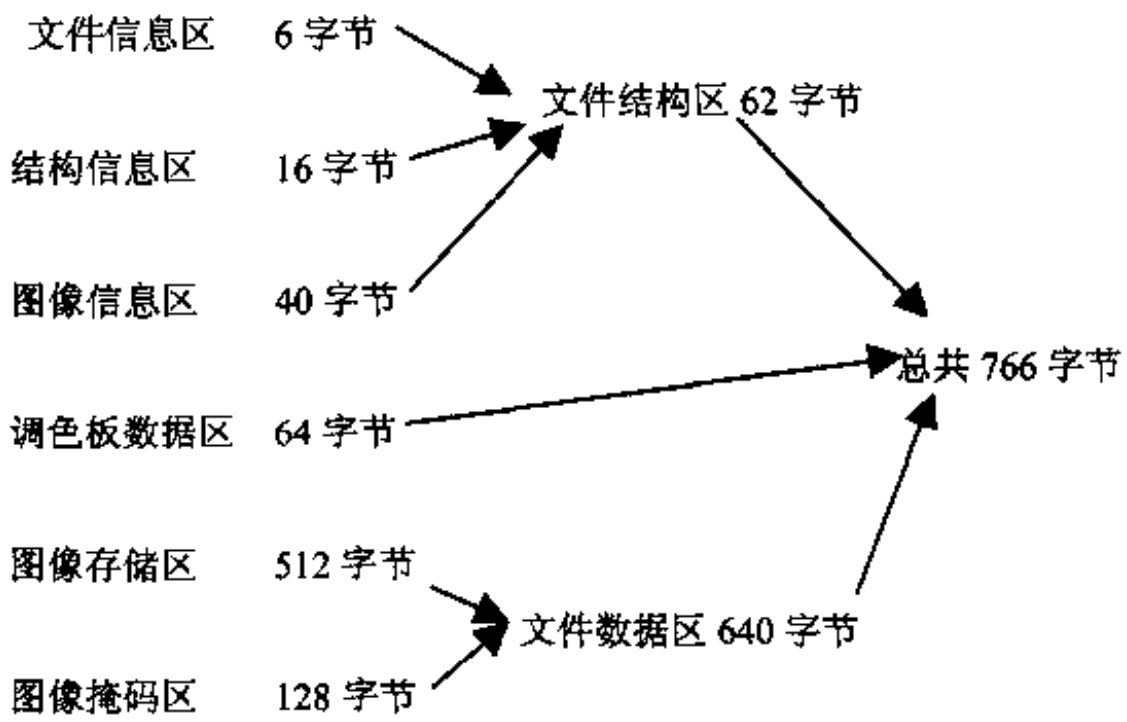


图 8-5　ico 文件数据分配

以下给出 ico 文件结构区的详细结构定义。

1. 文件信息区

```
struct Ico_file_head { //ico 文件头信息
    short uo_0,u2_1;  //U0,U2 通常分别为 0,1
    short image_count; //文件包含的图标个数
}*file_head;
```

我们来验证一下文件信息区的字节数量，TC 中 short 为 2 个字节，3 个 short 型变量为 3*2=6 个字节。

```
struct Ico_file_dir { //有关 ico 的结构信息
    char  width,height; //ico 尺寸，通常为 32*32(像素)
    short num_colors; //ico 使用的颜色数(16 色)
    short u4_0,u6_0; //通常为 0
    long  ico_size; //ico 图标信息长度(744Bytes)
    long  icon_offset; //ico 图标信息在文件中
}*file_dir;
```

再来验证一下文件信息区的字节数量，2 个 char 型、3 个 short 型、2 个 long 型，2*1+3*2+2*4=16 个字节。

```
struct Ico_head { //ico 头信息
long head_size; //通常固定值为 40
long width,height; //通常分别为 32，0
short planes; //通常为 1
short bits_per_pixel; //每个像素所占比特数,16 色模式下为 4
long compression; //ico 是否压缩
long image_size; //ico 图像数据长度
short reserved[8]; //保留值，全为 0
}*ico_head;
```

最后验证一下文件信息区的字节数量，10 个 short 型、4 个 long 型，10*2+5*4=40 个字节。以下给出 ico 文件调色板区结构：

```
typedef struct RGB_ico_typ {
unsigned char blue;
unsigned char green;
unsigned char red;
unsigned char reserved;
}RGB_ico,*RGB_ico_ptr;
```

ico 文件的调色板区和 bmp 文件的调色板区都有灰度，因此必须将调色板结构定义成

B、G、R、灰度，这是由程序一次完整读取方式所决定的（和 bmp 的对应读取不同，这将在稍后详细讲解）。前面我们介绍 256 色的 bmp 文件调色板区占用字节数为 256*4=1024 字节，而 ico 文件是 16 色的所以占用的字节数为 16*4=64 字节。

在建立了 ico 文件各部分结构的基础上，我们建立了一下的 ico 文件结构体：

```
typedef struct ico picture typ {
    file_head fhead;//文件信息区
    file_dir fdir;//结构信息区
    ico_head ihead;//图像信息区
    RGB_ico palette[16];//16 色调色板数据
    unsigned char map[512];//32*32 数据区，每个字节存放 2 个点颜色
    unsigned char mask[128];//32*32 掩码区，每个位对应一个点的掩码
    char far *buffer;//最终图像数据区指针
    } ico_picture, *ico_picture_ptr;
```

事实上这里的定义有一点重复，因为 map 和 mask[]数组变量本来就是描述 ico 图像区的，而我们还增加了一个指向将会被申请的 ico 内存空间的指针 buffer。那么本来已经用数组申请了空间还要申请空间是不是浪费呢？事实上在我们稍后了解了 ico 文件图像数据区是如何存放 32*32 个点阵的就会知道前后两者还是有一定区别的。

另外我们和先前 bmp 文件、pcx 文件一样使用 buffer 指针变量指向将被申请 ico 图像内存空间的做法也可以定义一个固定的数组来实现，这是由于 ico 文件点阵数量非常小只有 32*32=1024 个，不会占用很多空间。结构也可以定义如下：

```
typedef struct ico_picture_typ2 {
    file_head fhead;
    file_dir fdir;
    ico_head ihead;
    RGB_ico palette[16];
    unsigned char map[512];
    unsigned char mask[128];
    unsigned char buffer[1024];//存放最终每个点的颜色，一个字节一个点，32*32
    } ico_picture, *ico_picture_ptr;
```

这样我们就不需要在通过内存方式调用 ico 文件时为其申请和释放内存空间了。下面将详细介绍 16 色 32*32 点阵的 ico 文件数据区如何描述 32*32 点阵的。文件数据区包括图像数据区和图像掩码区。

2. 读取 ico 文件

以下是读取 ico 图像数据区的代码段：

```
for(i=32;i>0;i--)
    for(j=0;j<32;j=j+2) {
video_buffer[((y+i<<8)+(y+i<<6))+x+j]=ico.map[k]>>4;//读取第一个点
video_buffer[((y+i<<8)+(y+i<<6))+x+(j+1)]=ico.map[k++]&0x0f;//读取第二个点
```

图像数据区包括描述 16 色 32*32 点阵列的 512 个字节（要对应于 32*32 点阵颜色）。点阵有 32*32=1024 个点，1024 个点依靠 512 个字节表示，那么每个字节表示 2 个点，一个字节中的 4 位描述 1 个点，4 位正好为 16，也就是 16 色索引对应一个点。

在颜色阵列之后的是掩码阵列，也就是图像掩码区。所谓掩码就是透明码、屏蔽码，当 ico 文件某个点的掩码（一个点只需要 1 位做掩码）为 0 的时候就将该点输出到屏幕，如果为 1 就不用输出，换句话说该点在屏幕上是透明的。ico 文件中掩码阵列长度为 128 个字节（要对应于 32*32 点阵的透明性）。点阵有 32*32=1024 个点，1024 个点依靠 128 个字节进行掩码，那么每个字节掩 8 个点，一个字节中的 1 位正好掩 1 个点，1 位正好有 1（透明）和 0（掩）两种选择。16 色 32*32 点阵的 ico 文件显示图例见图 8-6。

这里还值得一提的是，对于 ico 文件调色板数据区的读取采用了和 bmp 文件不同的方式。虽然两种图像格式存放调色板数据都是以 B、G、R、灰度的顺序，但是在前面我们采用逐个读出三原色的 B、G、R 并对应给调色板结构体的 B、G、R 变量，而在 ico 文件的读取中将采用一次性全部原封不动全部读出到调色板数组空间的方法。读取到内存的代码段如下：

```
fread(&ico.palette,1,16*sizeof(struct RGB_ico_typ),fp); //一次性读取 16*4=64 个字节数据
```

然后对写入计算机调色板：

```
for(i=0;i<16;i++) Set_ICO_Palette_Register(i,(RGB_ico_ptr)&ico.palette[i]);
```

事实上文件调色板区数据在读取到调色板数组变量 palette 中的顺序还是 G、B、R、灰度，这就决定了我们必须将调色板结构体设置为同样的 G、B、R、灰度顺序。这与 bmp 调色板结构体设置的无顺序性是不同的。

再请与 bmp 文件调色板区读取对照一下，我们发现读取的每个数据移位工作都没有做，为了方便设置将计算机调色板进行如下修改：

```
void Set_ICO_Palette_Register(int index,RGB_ico_ptr color) {
outp(PALETTE_MASK,0xff);
outp(PALETTE_REGISTER_WR,index);
outp(PALETTE_DATA,color->red>>2);
outp(PALETTE_DATA,color->green>>2);
outp(PALETTE_DATA,color->blue>>2);
}
```

8.3.2　ico 文件显示

ico 文件的显示在解决了图像数据区、掩码数据区和调色板数据区的读取之后就显得非

常简单了。因为 ico 文件的显示过程和 bmp、pcx 文件非常类似。这里只讲解有所不同的读取 ico 文件的函数：

```
void ICO_Load(char *name,ico_picture_ptr ico) {
    unsigned char r,g,b;
    int BK = 256,k=0,i,j,x=0,y=0;
    FILE *fp;

    fp = fopen(name,"rb");//打开 ico 文件
//读取头信息
    fread(&ico->fhead,1,sizeof(struct Ico_file_head),fp);
    fread(&ico->fdir,1,sizeof(struct Ico_file_dir),fp);
    fread(&ico->ihead,1,sizeof(struct Ico_head),fp);
    fread(&ico->palette,1,16*sizeof(struct RGB_ico_typ),fp); //读取调色板信息
//读取数据区信息
    fread(&ico->map,1,512,fp);
    fread(&ico->mask,1,128,fp);
    fclose(fp);

    for(i=0;i<16;i++) {//设置计算机调色板
    Set_ICO_Palette_Register(i,(RGB_ico_ptr)&ico->palette[i]);
    }
//将 ico 数据区读入 buffer 内存
    for(i=31;i>=0;i--)
        for(j=0;j<32;j=j+2) {
            ico->buffer[i*32+j]=ico->map[k]>>4;
            ico->buffer[i*32+(j+1)]=ico->map[k++]&0x0f;
        }
//通过掩码判断修改 buffer 内存，获得最终 ico 图像数据
for(i=124;i>=0;i=i-4) {
    for(j=i;j<=i+3;j++) {
        unsigned char m=0x80;
        for(k=0;k<8;k++) {
            if(ico->mask[j]&m)//如果该位存在掩码
                ico->buffer[y*32+x]=BLACK;//这里设置黑色，应该是屏幕位置颜色
            x++;
            m>>=1;
        } }
    y++;
```

```
            x=0;
        } fclose(fp);
    }
```

函数使用了 6 个 fread 将 6 大部分内容全部读出，这是一种非常简练的方法（读出后再处理）。此后顺序完成设置调色板、将图像数据以点为单位写入申请内存以及根据掩码修改点阵内存。

以下给出的程序简化了掩码检验，这是由于其先读取并显示颜色区而后操作掩码区而造成的。仅将透明部分设为黑色。事实上透明部分应该是当前屏幕点的颜色（并不一定是黑色）。如果要让 ico 文件变得透明可以通过对 ico 文件颜色区、掩码区和屏幕对应点同时读取来实现，下面给出方法（以下程序没有使用这种方法）：

（1）先读当前点的掩码区；

（2）读取当前点的颜色区，如果刚才掩码区读出 1（透明）则不将颜色点显示到屏幕对应点，否则显示到屏幕对应点；

（3）循环第 1、2 步骤，直到结束。

通过内存显示 ico 文件的程序 icomem.c 与直接显示 ico 文件到屏幕的程序 ico.c 请查阅所附光盘的“source\8”目录。1.ico 文件如图 8-6 所示。

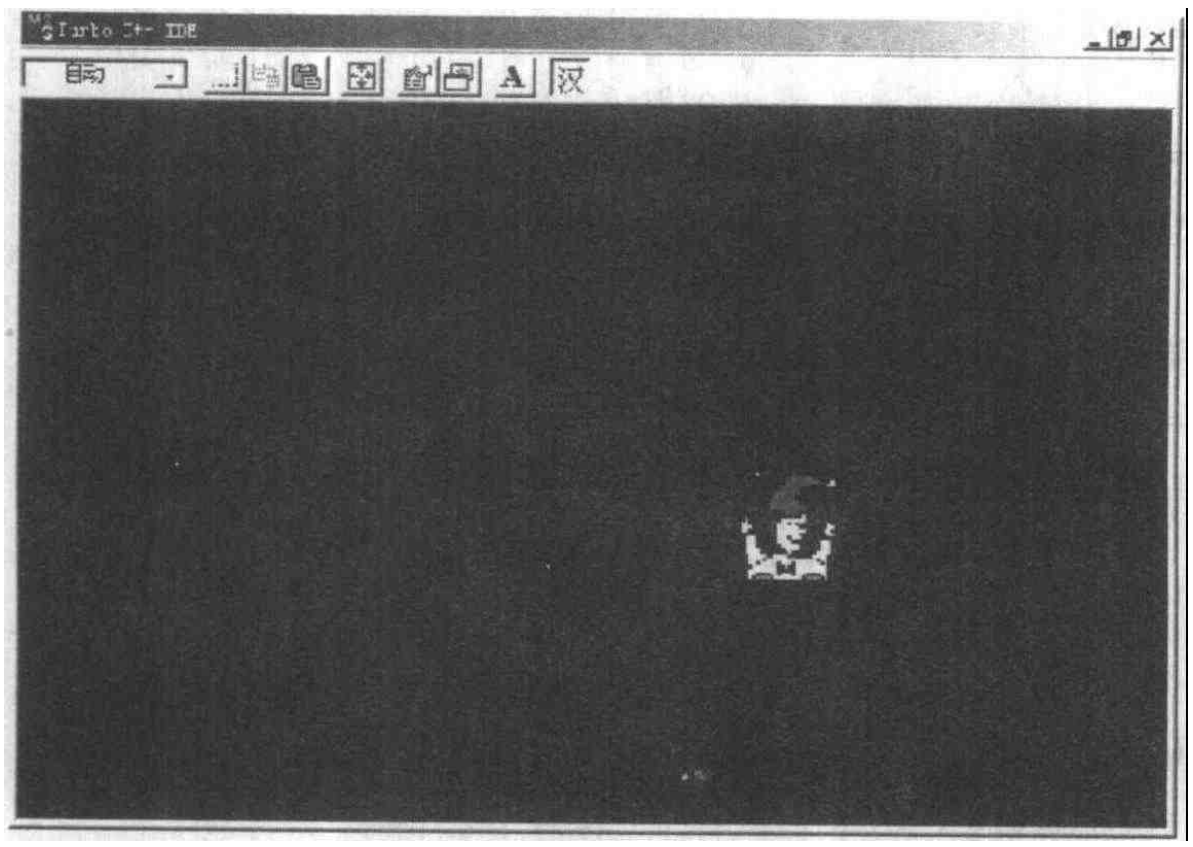


图 8-6　ico 文件显示示意

将许多 ico 文件放在一起是一件很有趣的事情，一个 ico 文件的动画（改进之后就可以做老虎机了）程序 icos.c 请查阅所附光盘的“source\8”目录。效果如图 8-7 所示。

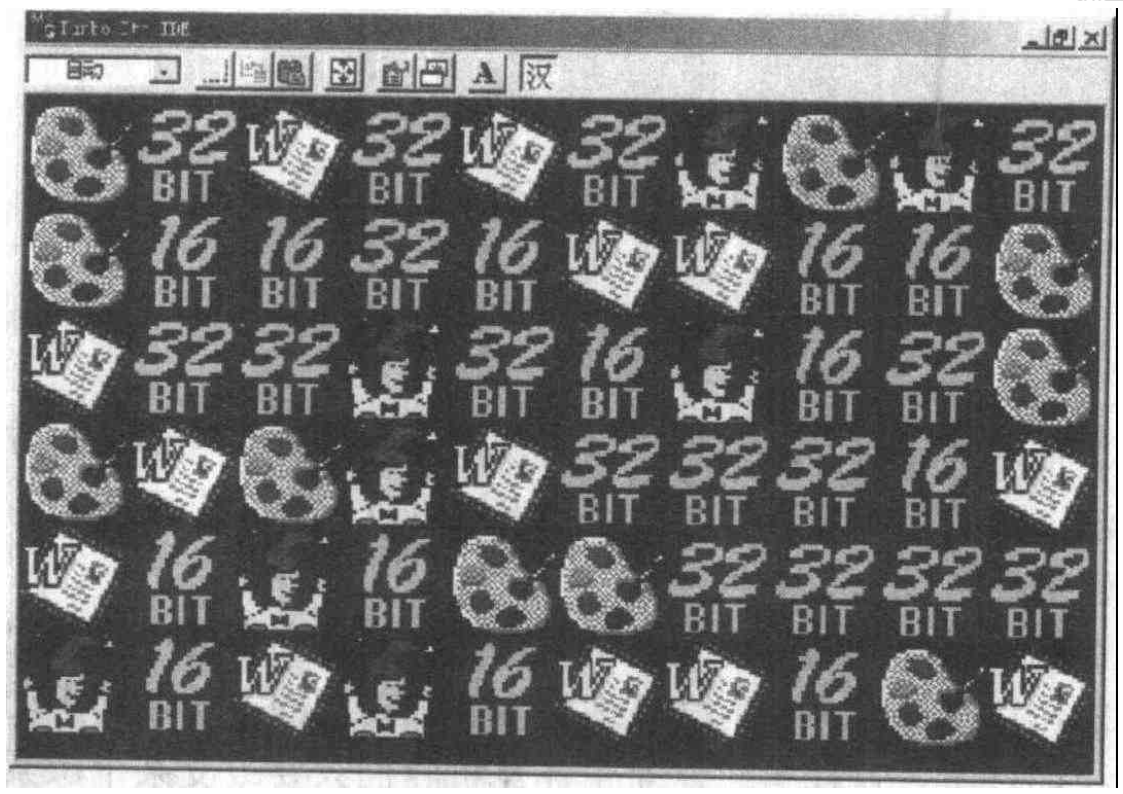


图 8-7　类似老虎机的 ico 图像示意

8.4　本 章 小 结

二维图形绘制和图形文件调用是组成游戏画面最常用的手段，其中尤以图形文件更为常用。学会了图形文件的调用我们就可以更加自如地设计游戏画面和游戏主人公形象，因为可以直接调取自己绘制的图像。以下是图像文件的程序调用过程：

（1）读取图像文件的文件头信息（包括长宽、分辨率等）；

（2）读取图像文件调色板信息，并且设置计算机调色板寄存器；

（3）读取图像文件图像数据到内存，有些需要进行解码处理；

（4）将读取的图像数据通过直接画点显示到屏幕对应位置。

C 语言编程最常用的三种图形文件是 bmp、pcx 和 ico 文件。三种图形文件的格式和应用范围各不相同。bmp 文件制作最为简单，我们只需要使用画笔就可以完成，但它是位图文件，所以比较大。pcx 文件的图像压缩率高且压缩方式简单，是比较流行的游戏编程图像。ico 文件是 Windows 中的图标文件，其形状规则且通常都是 32*32 像素，有大量的固有资源，使用起来比较灵活。

由于要一次性申请 320*200 像素 256 色图像的常规内存空间比较困难，我们使用直接写屏方式将所有图像写入屏幕显存。对于需要反复使用的图像（比如后面的子画面技术）我们可以将其中部分画面保存到常规内存中。对于许多张图片一次性读取的问题（游戏地图）我们只能依靠后面介绍内存技术实现了。

学后建议

（1）改写 ico 文件透明部分（先前的函数对透明部分填充黑色）显示的函数；

（2）研究 bmp、pcx、ico 图形文件更高的分辨率和像素情况下的存放规则，编制更加通用的图形文件读取和显示函数；

（3）研究 gif 图形文件和 fil 动画文件的存放格式，并且编制对应的读取和显示函数。

第9章　动 画 原 理

本章导读

第 3 章介绍了简单的动画实现技术，当时初步提到三种实现动画的方法：幻灯片、局部重画和异或方法。事实上幻灯片和局部重画方法可以归类于重画技术。而真正的电脑动画实现办法还有好多种。

对于动画的理解，通常我们认为是由于对象的移动而产生的。这种观点完全正确，然而换一个角度去看待这个问题，我们会发现更多实现动画的办法。其实动画也可以被看作固定位置颜色变化的结果。这样一来，对于计算机我们只要改变某个像素的颜色就可以实现动画了。改变像素颜色的办法并非通常以为的改变显存对应位置数据这一种，我们还可以通过在显存和显示器中间的颜色解释装置——调色板寄存器。比如显存中对应第一个屏幕点的数据是 00001111b=15 号颜色，通常这个数据对应的颜色是白色，然而如果将 00001111b 调色板寄存器的对应颜色改成其它颜色（比如黑色），这样也改变了屏幕像素颜色。

以上的例子只是希望告诉大家，如果充分发挥自己的想象力和计算机方面的知识就可能想出更多实现动画的办法或者可以有更多意想不到的途径来制作丰富多彩的游戏。

本章将系统地介绍实现动画的各类技术。

本章重点

（1）电脑动画的实现技术：重画技术、异或技术、调色板技术、拉屏技术，及其程序实现办法；

（2）计算机最常用的动画技术——重画技术及其分类：直接写屏、缓冲技术，及其程序实现办法；

（3）调色板技术实现动画的办法，通过改变调色板寄存器中的每种颜色的数据来实现，而不需要改变屏幕中画面图像。

9.1　动画技术分类

计算机动画技术从理论上说是非常多的。对于 C 语言编程比较适合的主要包括：

（1）重画技术：画了擦，擦了再画，或者一张一张贴（覆盖）上去；

（2）异或技术：通过对写入点和屏幕点颜色进行逻辑异或运算，来实现擦除和重画运动部分的动画；

（3）调色板技术：利用预先设置的动作图片和显示适配其中彩色表（以后称调色板寄存器）通过屏幕的颜色变化来实现动画的技术；

（4）拉屏技术：通过对保存画面的内存和屏幕显存内容的大幅度偏移复制（通常用于游戏中整屏地图的移动）实现的屏幕整体移动效果。

这之中重画技术是最为重要和流行的一种。重画技术又可分为：

（1）直接重画：通过画，清屏，重画，缺点容易产生闪动；

（2）缓冲：通过开辟一个对应于显存的缓冲内存空间，将所有绘制操作针对缓冲（后台绘制），绘制完成后复制到显存；

（3）页替换：有的图形模式下显存有两个显存分页（否则可以开辟两个缓冲做弥补），将屏幕首地址交替指向两个显存页（如果是开辟两个缓冲，则是交替复制给显存），并且对当前非显示页进行绘制的动画技术；

（4）多缓冲：为了使动画画面更加稳定和流畅，在双缓冲技术上开辟更多的对应显存的缓冲空间，建立缓冲顺序依次进行后台同步绘制，从而保证不会出现屏幕等待情况。

以下将介绍各类技术的原理和具体实现方法。

9.2　重 画 技 术

重画技术简单的理解就是画了擦，擦了再画。重画的过程一般如图 9-1 所示。

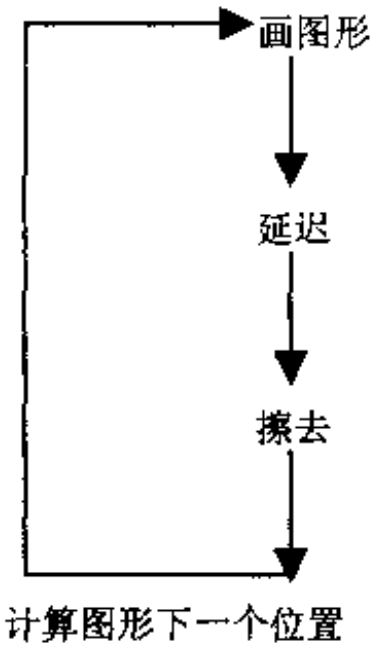


图 9-1　实现重画示意

重画技术中包括直接重画，双缓冲和多缓冲技术。其中文件重画动画是直接重画的一个应用。双缓冲技术经常应用在工作量较大的子画面重画中。

9.2.1　直接重画

所谓直接重画，就是不断直接重画屏幕显存。在之前的章节中介绍文本的显存为 B8000H，而图形显存为 A0000H。所以直接重画就是所有重画都只针对于 A0000H 显存。

在第 3 章简单动画中运动的圆那个例程就是属于直接重画。它将圆画到屏幕以后保留了很短时间然后擦除，接着重新计算圆的位置并再将它画到屏幕。在那个例程中使用到了重画技术的两种小分类：全屏重画和局部重画。

全屏重画就是每次在画面绘制并停留之后进行清屏，而局部重画只重画屏幕画面中运动图像的区域，画面中静止的图像并不重画。

全屏重画通常使用于：

（1）动画画面较大；

（2）动画元素太多；

（3）动画元素运动区域不明确。

局部重画通常使用于：

（1）静止元素较多并且绘制工作量较大；

（2）动画元素较少并且绘制工作量较小。

文件重画是直接重画的一个典型的应用，它属于全屏重画。所谓文件重画是指将事先画好的图片文件序列通过直接重画的方法依次放到屏幕上而产生动画效果。

上一章提到游戏中最常用的图形格式是 pcx 文件。bmp 文件动画在上一章已经给出了例程，但是由于其文件占用空间太大，不适于在游戏中使用。这里将介绍用 pcx 文件序列实现动画的方法。

具体的制作方法如下：

（1）用 Windows 画笔、photoshop 和 photoimpact 画一个基本图画；

（2）然后使用复制、绘画等方法对其进行渐变操作，每一种变化保存到一个图像文件中；

（3）使用 acdsee 软件将其转化为 pcx 格式；

（4）将 pcx 图像文件序列名加入显示程序中，同时将显示程序改成一个循环显示语句，每次延迟足够时间。

9.2.2　缓冲技术

所谓缓冲技术是针对直接写屏（直接重画）而言的。我们采用直接写屏的时候总是在前一个画面清屏后在屏幕上图图画画，这种将操作放在观众的面前的做法不仅浪费了很多时间更会导致屏幕的闪动。于是我们考虑将操作放到幕后，待操作完成后再将它快速映射到屏幕上去。事实上这将大大提高动画显示的速度和稳定性。

双缓冲技术要求我们开辟一块和我们要显示对象相同大小（不一定和显存一样大）的内存区域。每次操作都只对这块新开辟的内存进行，而在屏幕上我们只可能看到完工了的画面。与直接写屏不同的是，我们要多完成开辟内存区域和映射内存到显存两部分任务。

缓冲技术具体的过程可以被理解为：有两个屏幕，一个是当前屏幕、一个是绘制屏幕。当前屏幕实际上就是我们可以直接看到的屏幕（显存），而绘制屏幕就是我们开设的与显存大小相同的内存区域。我们将一张画面绘制到绘制屏幕（申请的内存）中去，等当前屏幕内容显示了足够的时间后再将其替换上去。所以，当前屏幕上从未进行过任何绘制操作，所有操作都在绘制屏幕上完成。

双缓冲技术在具体的使用中要注意以下几点：

（1）通常比较难申请到与屏幕相同大小（320*200 字节）的缓冲内存；

（2）如果只有单个小区域动画画面，可以针对该区域申请缓冲；

（3）如果出现多个区域动画，不建议为每个区域申请一个相同大小的缓冲，而是建议使用一个较大的缓冲区域。

建立缓冲函数如下：

```
int Create_Double_Buffer(int num_lines) { //申请缓冲空间
if ((double_buffer=(unsigned char far*)malloc(SCREEN_WIDTH*(num_lines+1)))==NULL)
return(0);
```

```
buffer_height=num_lines;
buffer_size=SCREEN_WIDTH*num_lines/2;
_fmemset(double_buffer,0,SCREEN_WIDTH*num_lines);//缓冲内清 0
return(1);
}
```

程序中释放双缓冲函数如下：

```
void Delete_Double_Buffer(void) {
if(double_buffer)
free(double_buffer);//释放缓冲空间
}
```

程序中映射缓冲函数如下：

```
void Show_Double_Buffer(char far *buffer) {
_asm{
push ds
mov cx,buffer_size
les di,video_buffer
lds si,buffer
cld
rep movsw
pop ds
} }
```

这里采用在 C 语言中调用汇编语言块复制语句的高效做法。以下给出一个直接写屏程序和一个缓冲程序。

1. 直接写屏

程序功能：通过直接写屏的方法在屏幕上不断随机产生点、线、圆等图形。

程序流程：

（1）初始化图形模式；

（2）直接画多种图形；

（3）循环第 2 步骤，直到有键盘按键退出图形模式。

主要函数：

```
void Bline(int x0,int y0,int x1,int y1,unsigned char color); //画任意直线
void Circle(int x,int y,int r,int color); //画圆
void Create_Tables(void); //建立速查表
void Fill_Screen(int value); //屏幕填充
void H_Line(int x1,int x2,int y,unsigned int color); //画横线
```

```
void V_Line(int y1,int y2,int x,unsigned int color); //画竖线
void Delay(int clicks); //时间延迟
void Set_Video_Mode(int mode); '/设置显示模式
void Plot_Pixel_Fast(int x,int y,char color); //画点
```

程序要点：所有图形绘制操作都是对显存进行的。

程序代码 nobuffer.c 请查阅所附光盘的“source\9”目录。

2. 缓冲

程序功能：通过对申请的内存空间进行不断随机产生点、线、圆的操作和将内存数据复制给显存的方法来显时图形。

程序流程：

（1）申请内存空间；

（2）初始化图形模式；

（3）向内存空间画多种图形；

（4）将内存数据复制到显存；

（5）循环 3、4 步骤，直到有键盘按键退出图形模式。

主要函数：

```
void Bline_DB(int x0,int y0,int x1,int y1,unsigned char color); //向缓冲画任意直线
void Circle_DB(int x,int y,int r,int color); //向缓冲画圆
void H_Line_DB(int x1,int x2,int y,unsigned int color); //向缓冲画横线
void V_Line_DB(int y1,int y2,int x,unsigned int color); //向缓冲画竖线
void Plot_Pixel_Fast_DB(int x,int y,char color); //向缓冲画点
void Show_Double_Buffer(char far *buffer); //显示内存空间
```

程序要点：所有图形绘制操作都是对申请的内存进行，然后统一复制给显存。

程序代码 buffer.c 请查阅所附光盘的“source\9”目录。

9.3　异 或 技 术

异或技术已经在第 3 章中进行过详细介绍。异或在数学中的公式是：

A 异或 B=A*$\overline{B}$+$\overline{A}$*B

它在图形显示中的意义：如果在同一个位置用异或的方法画两个颜色相同的点，点会消失；而用不同颜色画点，点会变成第三种颜色。这里将给出一些自己编写的基本图形异或函数。

1. 异或点

```
void Plot_Pixel_Fast_Xor(int x,int y,char color) {
int color_screen;
color_screen=Get_Pixel(x,y);//取屏幕点
```

```
video_buffer[((y<<8)+(y<<6))+x]-color^color_screen;//写入点与屏幕点异或
}
```

事实上由于所有图形都是基于点绘制而成的，所以我们只要了解如何绘制一个异或点就可以实现各类图形的异或函数了。点异或其实就是将当前要放入屏幕的颜色和屏幕当前点位置的颜色进行异或操作，然后再复制到屏幕上的行为。首先我们从屏幕当前位置取得点的颜色，然后使用 C 语言给我们提供了位异或操作的操作符“^”，可以方便的实现屏幕点和绘制点之间的颜色异或操作。

2. 异或横线

```
void H_Line_Xor(int x1,int x2,int y,unsigned int color) {
    int i;
    int color_screen;
    for(i=x1;i<x2;i++) {
        color_screen=Get_Pixel(i,y);
        video_buffer[ (y<<8) + (y<<6) + i]=color^color_screen;
    } }
```

3. 异或竖线

```
void V_Line_Xor(int y1,int y2,int x,unsigned int color) {
    int i;
    int color_screen;
    for(i=y1;i<y2;i++) {
        color_screen=Get_Pixel(x,i);
        video_buffer[ (i<<8) + (i<<6) + x]=color^color_screen;
    } }
```

4. 异或正方形

```
void Square_Xor(int x,int y,int side,int color) {
    H_Line_Xor(x,x+side,y,color);
    H_Line_Xor(x,x+side,y+side,color);
    V_Line_Xor(y,y+side,x,color);
    V_Line_Xor(y,y+side,x+side,color);
}
```

5. 异或填充正方形

```
void Fill_Square_Xor(int x,int y,int side,int color) {
    int i;
    for(i=y;i<=y+side;i++) {
        H_Line_Xor(x,y+side,i,color);
```

```
    } }
```

6. 异或矩形

```
void Rectangle_Xor(int x1,int y1,int x2,int y2,int color) {
    H_Line_Xor(x1,x2,y1,color);
    H_Line_Xor(x1,x2,y2,color);
    V_Line_Xor(y1,y2,x1,color);
    V_Line_Xor(y1,y2,x2,color);
}
```

7. 异或填充矩形

```
void Fill_Rectangle_Xor(int x1,int y1,int x2,int y2,int color) {
    int i;
    for(i=y1;i<=y2;i++) {
        H_Line_Xor(x1,x2,i,color);
    } }
```

8. 异或圆

```
void Circle_Xor(int x,int y,int r,int color) {
    int x0,y0,x1,y1,index;
    x0=y0=r/sqrt(2);
    for(index=0;index<=360;index++) {
        x1=x0*cos_look[index]-y0*sin_look[index];
        y1=x0*sin_look[index]+y0*cos_look[index];
        Plot_Pixel_Fast Xor(x+x1,y+y1,color);
    } }
```

一个简单的异或图形例程 xor.c 请查阅所附光盘的“source\9”目录。

9.4 调色板技术

9.4.1 调色板寄存器

有没有玩过 street frighter（街霸）？你是否注意到在打斗双方后面的背景中的灯光会发生时暗时亮的颜色变化和只有两三个循环动作的加油者？对了，这个就可以用调色板动画技术来实现。

在讨论调色板动画技术之前，我们首先要了解一下计算机中的调色板。

我们采用的图形模式 13h 提供了 256 种颜色。这些颜色都是来自于一个调色板寄存器。如同一个画家决定画一副 256 色油画作品之前，首先要在调色板中调出 256 种颜色，并且每种颜色放在调色板的一个格子空间中，事实上这个调色板一共也就是由 256 个格子组成。当画家要使用某种颜色的时候，他只需要根据这种颜色在调色板中的序号将其找出便能够

画到画布上去了。

计算机调色板中每一个格子也被放入了一种颜色。同时每种颜色都是依靠红、绿、蓝三原色的不同比例混合而成的。现在的问题非常简单：

（1）计算机调色板的基本结构到底是怎么样的；

（2）如何从计算机调色板中读取调色板序列颜色；

（3）如何将我们自身设定的调色板序列颜色放入计算机调色板中。

首先来了解一下计算机调色板寄存器的基本结构。

计算机调色板寄存器的调用硬件端口是 3C6H，读端口是 3C7H，写端口是 3C8H，数据端口从 3C9H 开始。调色板共有 256 个色彩单位，每个色彩单位又分红、绿、蓝三原色，每种基本颜色的色彩范围是 0~255，三个值结合起来配出需要的颜色放入调色板寄存器中。那么我们就可以通过写硬件端口 outp(portid，value)来操作调色板。为了方便我们定义了一个调色板色彩单位结构体如下：

```
typedef struct RGB_color_typ {
unsigned char red;
unsigned char green;
unsigned char blue;
}RGB_color,*RGB_color_ptr;
```

可以看出：

（1）调色板寄存器 256 个色彩单位都是由三原色结构配色而成的。而“unsigned char red;”说明基色红颜色是由一个 8 位的二进制变量来表示的，红色取值的确是从 0~255，并且数值越大颜色越红。同理，绿色和蓝色也都是如此的。

（2）在计算机调色板寄存器的每个色彩单位中，三原色永远是以红、绿、蓝的顺序排列的，这在写入和读取调色板寄存器的时候是相当需要重视的。

其次来了解一下读取计算机调色板寄存器的方法。

由于我们在写游戏的时候通常要改变调色板寄存器，于是在改变之前进行备份是非常重要的。读取调色板寄存器的过程非常简单，以读取调色板中第 index 号颜色为例顺序如下：

（1）向计算机调色板寄存器调用硬件端口 3C6H 写入 ffH，启动调色板寄存器调用功能；

（2）向调色板寄存器读端口 3C7H 写入要读取的颜色序号 index；

（3）通过由调色板色彩单位结构体 RGB_color_typ 定义的指针型变量 color 依次（3 次）从读端口 3C7H 取出红、绿、蓝三种基色使用量。

以下给出取调色板某一索引颜色的函数：

```
#define PALETTE_MASK 0x3c6
#define PALETTE_REGISTER_RD 0x3c7
#define PALETTE_REGISTER_WR 0x3c8
#define PALETTE_DATA 0x3c9
```

```
void Get_Palette Register(int index,RGB_color_ptr color) {
outp(PALETTE MASK,0xff);//启动调色板
outp(PALETTE_REGISTER_RD,index);//读调色板 index 序号内寄存器
color->red=inp(PALETTE_DATA);//读红色数据
color->green=inp(PALETTE_DATA);//读绿色数据
color->blue-inp(PALETTE_DATA);//读蓝色数据
}
```

如果要取出调色板寄存器所有 256 种颜色，只需要使用 for 循环语句对该函数进行重复调用即可，如下：

```
int index;
   RGB_color_ptr color[256];
   ...
   ...
   for(index=0;index<256;index++)
       Get_Palette_Register(index,color[256]);
```

最后介绍一下向计算机调色板寄存器写入调色板序列的方法。

在改变调色板寄存器或者恢复原先备份的调色板寄存器内容的时候我们都要使用到协调色板寄存器。其过程与读取调色板寄存器的过程非常类似：

（1）向计算机调色板寄存器调用硬件端口 3C6H 写入 ffH，启动调色板寄存器调用功能；

（2）向调色板寄存器写端口 3C8H 写入要写入的颜色序号 index；

（3）通过由调色板色彩单位结构体 RGB_color_typ 定义的指针型变量 color 依次（3 次）向读端口 3C8H 红、绿、蓝三种基色赋值。

以下给出写调色板某一索引颜色的函数（在前一章图像文件显示中已经使用过了）：

```
void Set_Palette_Register(int index,RGB_color_ptr color) {
outp(PALETTE_MASK,0xff);//启动调色板
outp(PALETTE_REGISTER_WR,index);//写调色板 index 序号
outp(PALETTE_DATA,color->red);//写入红色数据
outp(PALETTE_DATA,color->green);//写入绿色数据
outp(PALETTE_DATA,color->blue);//写入蓝色数据
}
```

要向调色板寄存器写入所有 256 种颜色，也只需要使用 for 循环语句对该函数进行重复调用即可，如下：

```
for(index=0;index<256;index--)
    Set_Palette_Register(index,color[256]);
```

9.4.2　调色板动画原理

所谓调色板动画技术就是，我们画了一个点之后不用擦掉它或是重画，而通过改变它所用颜色的调色板中色彩单位中的颜色来改变这个点的颜色从而产生动画效果。例如用调色板寄存器中 1 号色彩单位（此时 1 号色彩单位内放的是绿色）内的颜色画了一棵树，于是树是绿色的；然后我们将 1 号色彩单位中的颜色改成红色，那么树立刻变成了红色了。

之所以能够产生这种效果其原因是我们可以通过任意地修改调色板寄存器色彩单位来改变屏幕颜色。这与画家用调色板画画有所不同，画家用调色板中 1 号格子中的颜色（比如是绿色）向画布上的树填上绿色后，决定改变调色板 1 号格子的颜色为红色，此时虽然 1 号格子中的颜色变成了红色，可是画布上的树依然是绿色。而计算机调色板寄存器某一色彩单位内颜色一旦发生变化，其屏幕对应位置（使用该色彩单位的点）将同时发生变化。

所以，调色板动画技术说到底就是改变颜色，是通过颜色的变化来产生动画效果的。

一个显示并改变计算机调色板寄存器 256 种颜色的程序 pal.c 请查阅所附光盘的“source\9”目录。调色板改变前后的 2 张图像如图 9-2 及图 9-3 所示。

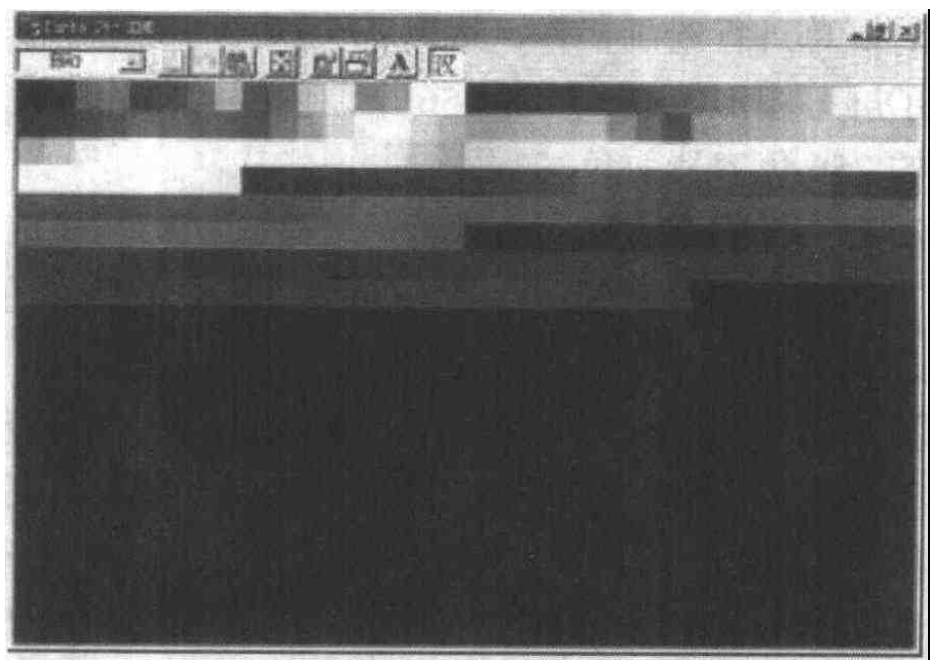


图 9-2　改变前

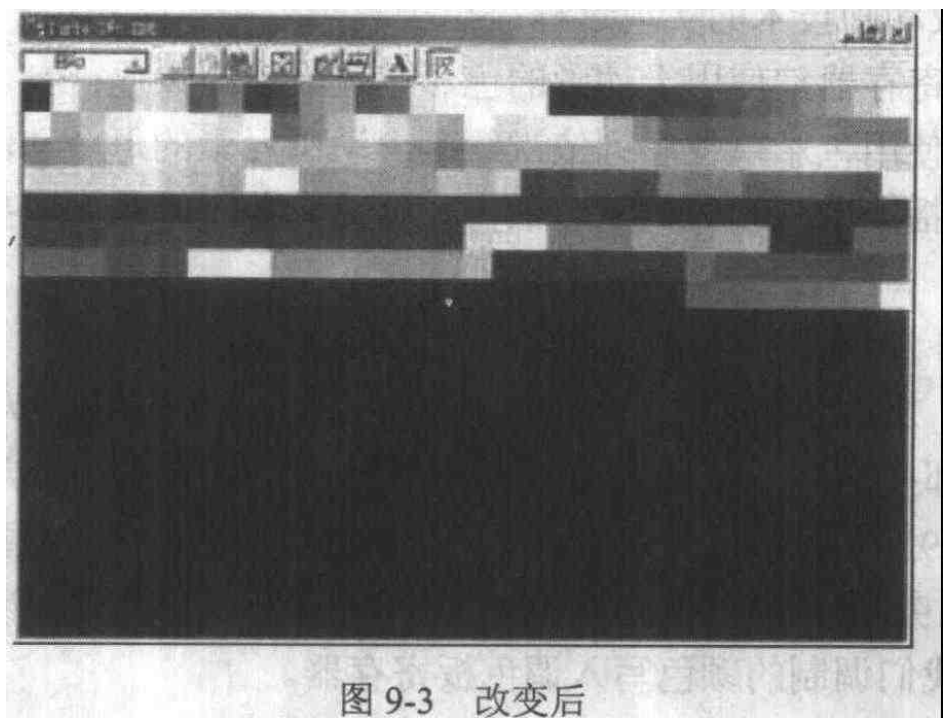


图 9-3　改变后

此程序首先通过写点函数将计算机调色板寄存器 256 个色彩单位颜色以色块的形式写入屏幕。然后通过循环读取调色板寄存器函数读取每个色彩单位中的 R、G、B 值，并且分

别对三原色减 10（降低颜色程度），而后通过循环写入调色板寄存器函数将颜色放回调色板寄存器对应色彩单位。这样便产生了调色板发生颜色变化的效果。

程序中为了将所有的调色板色彩单位都设置成黑色，于是将：

```
color.red=color_1.red-10;
color.green=color_1.green-10;
color.blue=color_1.blue-10;
```

改为：

```
color.red-0;
color.green=0;
color.blue=0;
```

便实现了。

调色板动画技术的应用非常有特色。和其它动画技术最大的不同是：一般动画技术都是在动画播放过程中对屏幕进行绘画操作，通常都会改变显存内容。而调色板动画技术在动画播放之前就完成了屏幕图形绘制（所有 256 个调色板容器此时设置为黑色），而在动画过程中只改变调色板的 256 色容器颜色。

这种动画实现方法有一个非常大的优点，就是动画画面极其稳定、流畅。原因很简单，所有的动画元素的所有动画序列过程都已经在屏幕上了。而改变调色板颜色根本不对屏幕进行操作，更不会引起屏幕闪动。

不过调色板动画技术也有其缺陷，主要是无法处理复杂的动画元素。原因是所有绘制都在动画之前完成，于是所能绘制的范围只有 1 个屏幕的大小（一般动画在每次重画过程中都有 1 个屏幕可以绘制），要一次将动画所有序列画在屏幕几乎是不可能的。除非对于处理较为简单的动画元素或者只有重复两三个动作的动画元素（比如“街霸”游戏背景中只有两三个重复动作的加油人物）调色板动画技术可以大显身手。

以下给出调色板动画技术的实现过程：

（1）将调色板寄存器被使用于动画的色彩单位设置为黑色；

（2）在屏幕上使用色彩单位序列依次绘制出动画元素的每个动画动作；

（3）设置当前播放动画画面的调色板寄存器色彩单位颜色为需要显示的颜色；

（4）延迟；

（5）循环 3、4 步骤，直到所有动画画面结束或者游戏结束。

值得一提的是，256 个色彩单位并不一定在一开始就全部设置为黑色。我们可以将被使用于动画效果的那些色彩单位设置为黑色，而另一些则还可以作为屏幕上不变幻的图像颜色。比如，使用 192 个色彩单位作为动画效果，余下的 64 个作为图像颜色；这时候只需要在程序开始将前 192 个色彩单位设置为黑色，而后 64 个可以根据颜色要求或者使用现有的色彩或者通过将我们调制的颜色写入调色板寄存器。

9.4.3　调色板动画举例

以下给出四个调色板动画例程。

1. 清屏（屏幕渐暗）

程序功能：通过对计算机调色板寄存器所有色彩单位的三原色进行递减操作，使屏幕最终变为黑色，实现艺术清屏功能。

程序流程：

（1）将计算机调色板寄存器所有256种色彩单位显示于屏幕；

（2）依次从调色板寄存器读取每个色彩单位的R、G、B三原色；

（3）分别对三原色进行减1操作；

（4）将减1后的三原色写回刚才的色彩单位中去；

（5）循环2~4步骤，直到所有调色板寄存器的R、G、B三原色的值都变为0。

主要函数：

```
void Set_Palette_Register(int index,RGB_color_ptr color); //设置调色板颜色
void Get_Palette_Register(int index,RGB_color_ptr color);  //读取调色板颜色
void Delay(int clicks); //时间延迟
void Set_Video_Mode(int mode); //设置显示模式
void Plot_Pixel_Fast(int x,int y,char color); //画点
```

程序要点：当R、G、B三原色取值都为0的时候（取值范围0~255），调色板寄存器色彩单位所呈现的颜色为黑色。

程序代码black.c请查阅所附光盘的“source\9”目录。

2. 竖向扫动

程序功能：实现一条横穿屏幕的绿色横线从上向下在屏幕中扫动。

程序流程：

（1）将调色板寄存器前200个色彩单位设置为黑色；

（2）依次用调色板寄存器色彩单位序列前200号在屏幕上绘制200条横线；

（3）设置当前横线所使用的色彩单位为绿色，同时设置刚才为绿色的横线所使用的色彩单位为黑色；

（4）色彩单位序号加1（当前横线向下移动一条）；

（5）循环3、4步骤，直到横线在屏幕最下端的时候返回屏幕最上方的横线，按任意键退出。

主要函数：

```
void H_Line(int x1,int x2,int y,unsigned int color); //画横线
```

程序要点：

（1）调色板寄存器被使用于动画制作的色彩单位在绘制画面前就应该全部设置为黑色；

（2）要产生绿色横线向下移动效果，需设置当前色彩单位（横线）为绿色同时将刚才为绿色的色彩单位（上面一条横线）恢复为黑色；

（3）不要改变调色板寄存器0号色彩单位（0号为屏幕底色，通常为黑色），否则屏

幕底色将发生变化。

程序代码 colorrot.c 请查阅所附光盘的“source\9”目录。效果如图 9-4 所示。

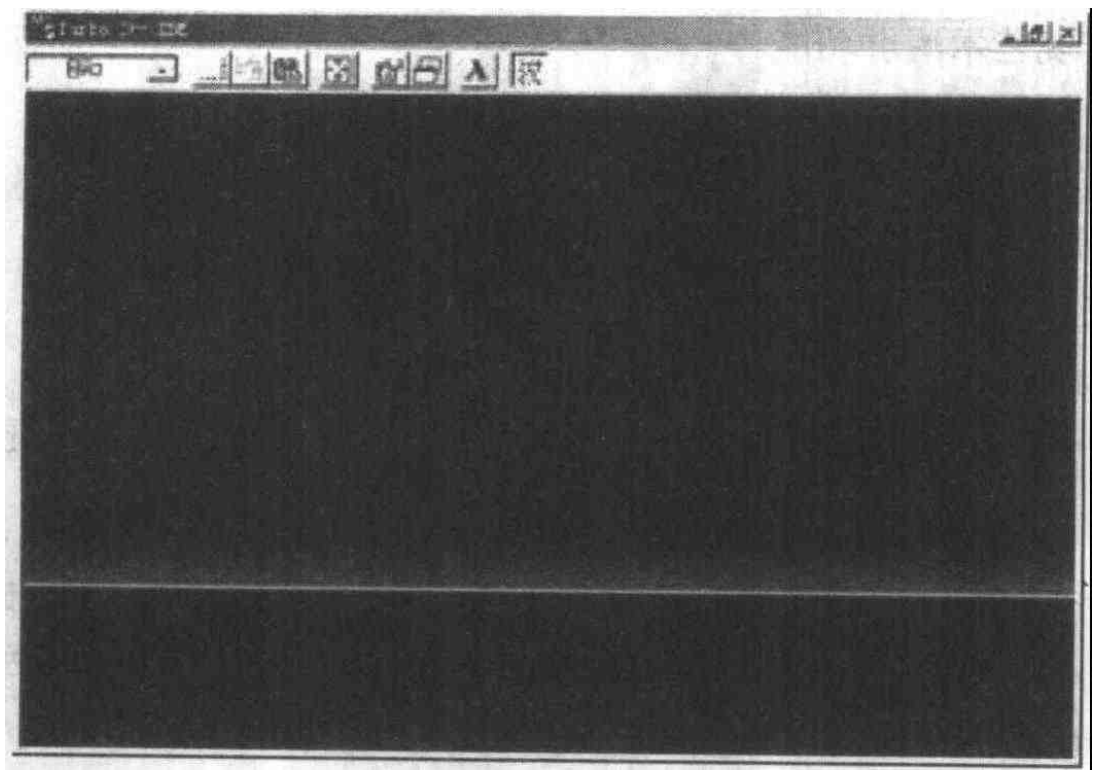


图 9-4　直线下移消隐

3. 雷达回扫

程序功能：这是我认为自己设计得相当经典的一个调色板程序。产生类似于红色警戒片头中的雷达回扫动画，绿色的扫描线在一个固定的半径中循环扫动，同时在雷达中出现红色敌机影像。

程序流程：

（1）将调色板寄存器所有色彩单位设置为黑色，然后初始化部分动画色彩单位颜色；

（2）绘制雷达框架，并使用任意直线函数绘制出一个固定出发点的全方向（360 度）直线（每 2 条直线使用 1 种色彩单位，共使用 180 个色彩单位），最后使用画点函数绘制敌机影像轨迹；

（3）设置当前雷达扫描线所使用的色彩单位为绿色，并拖出 15 条渐暗的扫描线于其后，设置当前敌机位置点所使用的色彩单位为红色，同时将上一个敌机位置点所使用的色彩单位设置为雷达屏幕的深绿色；

（4）循环第 3 步骤，直到键盘被敲击。

主要函数：

```
void Create_Tables(void); //建立角度 cos 表和 sin 表
void BPlot_Pixel_Fast(int x,int y,char color); //画粗点
void Bline(int x0,int y0,int x1,int y1,unsigned char color); //画任意直线
void V_Line(int y1,int y2,int x,unsigned int color); //画竖线
void Circle(int x,int y,int r,int color); //画圆
void BCircle(int x,int y,int r,int color); //画粗线条圆
void Fill_Circle(int x,int y,int r,int color); //绘制填充圆
```

程序要点：

（1）固定出发点的 360 度雷达扫描线的终点位置需要使用到角度 cos 值和 sin 值，具体的计算方法是初始化一条直线(x0,y0)到(x1,y1)，然后使用角度变换公式：

```
x1'=x0*cos(angle)-y0*sin(angle);
y1'=x0*sin(angle)+y0*cos(angle);
```

（2）由于 360 度雷达线无法遮盖圆形全部区域为深绿色，所以，在绘制扫描线之前先使用画圆函数用深绿色填充雷达区域；

程序代码 leida.c 请查阅所附光盘的“source\9”目录。其效果如图 9-5 及图 9-6 所示。

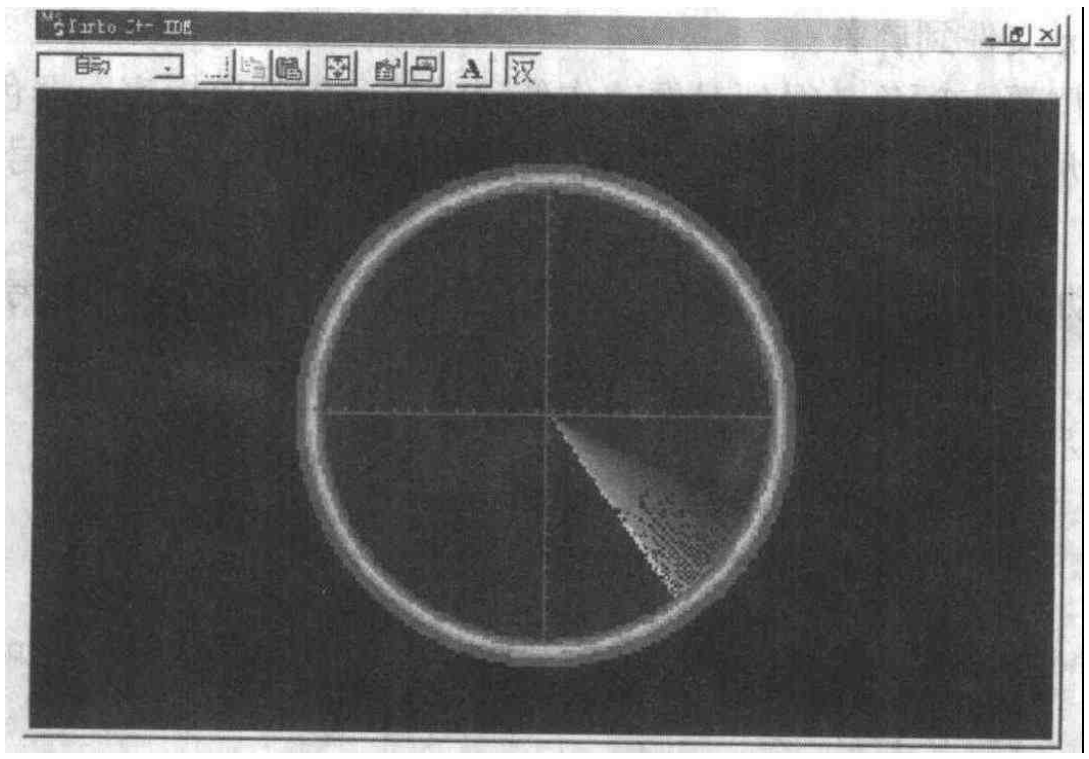


图 9-5　雷达扫描效果

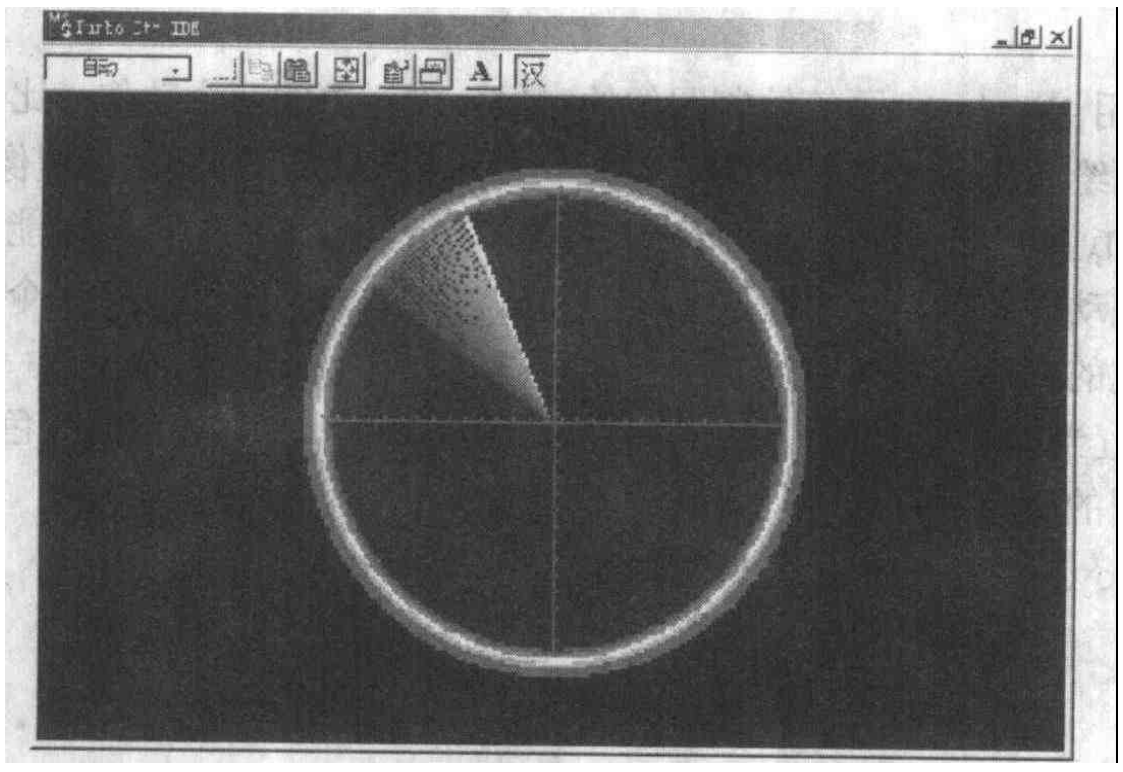


图 9-6　雷达扫描效果（进行中）

4. 对象重复动作动画

程序功能：调用一幅 pcx 图画（已绘制好 2 个动作），利用将动作 1 和动作 2 所使用的调色板寄存器色彩单位号码交替设置为动作颜色和背景颜色来实现动画效果。见图 9-7（第三幅为合成效果）。



图 9-7　pcx 插图及效果

程序流程：

（1）调入 pcx 文件到屏幕；

（2）通过取屏幕点颜色，得到对像动作使用到颜色的调色板寄存器色彩单位号码；

（3）将当前动作使用到的 2 个颜色所对应的调色板寄存器色彩单位号码设置为背景颜色；

（4）将另一动作使用到的 2 个颜色所对应的调色板寄存器色彩单位号码设置为动作颜色；

（5）延迟一点时间；

（6）循环第 3、4、5 步骤，直到按键退出。

主要函数：

```
void PCX_Load_Pcx(char *filename, pcx_picture_ptr image,int enable_palette); //将 pcx 文件直接写入显存
int Get Pixel(int x,int y); //取屏幕点的颜色（调色板寄存器色彩单位）
```

程序要点：

（1）在使用 Windows 画笔绘制图像的时候，将对象的两个动作（比如这里的手部运动）用不同颜色绘制出来（此时图像上有 4 个手），然后保存为 256 色图像；

（2）必须知道两个动作所涉及颜色对应的调色板寄存器色彩单位才能通过对其进行写入操作从而实现动画效果，要知道两个动作的颜色（比如手部运动有 4 个颜色）可以通过读取屏幕点颜色的办法实现；

（3）在设置当前动作 2 颜色的时候并不是设置其在 pcx 文件中的颜色，而是将其设置为和动作 1 相同的颜色，这样才能看出同一样东西在动；

程序代码 boy.c 请查阅所附光盘的“source\9”目录。

9.5　拉 屏 技 术

玩过玛丽兄弟么？请注意这个游戏中的环境屏幕始终是根据玛丽兄弟向前走而向左移。这里就是用到了拉屏技术。

拉屏技术事实上是对双缓冲技术的一个改进，它将原先需要全部 copy 的映射内存变为只要 copy 新移入屏幕的用于填补空白的一部分，而原先的屏幕内容在 copy 之前全部按照要求距离拉向了另一个方向。这样使大规模的内存间 copy 变成了内存自身的 copy 和小计量的内存间 copy，这大大提高了动画的速度和质量。

此外，在这种模式下的显存大出屏幕一些，于是拉屏技术可以得到更多的发挥了。具体方法如图 9-8。

（1）直接移动显存就可以实现动画；

（2）显存空白的部分通过双缓冲对应位置来补充。见图 9-8。

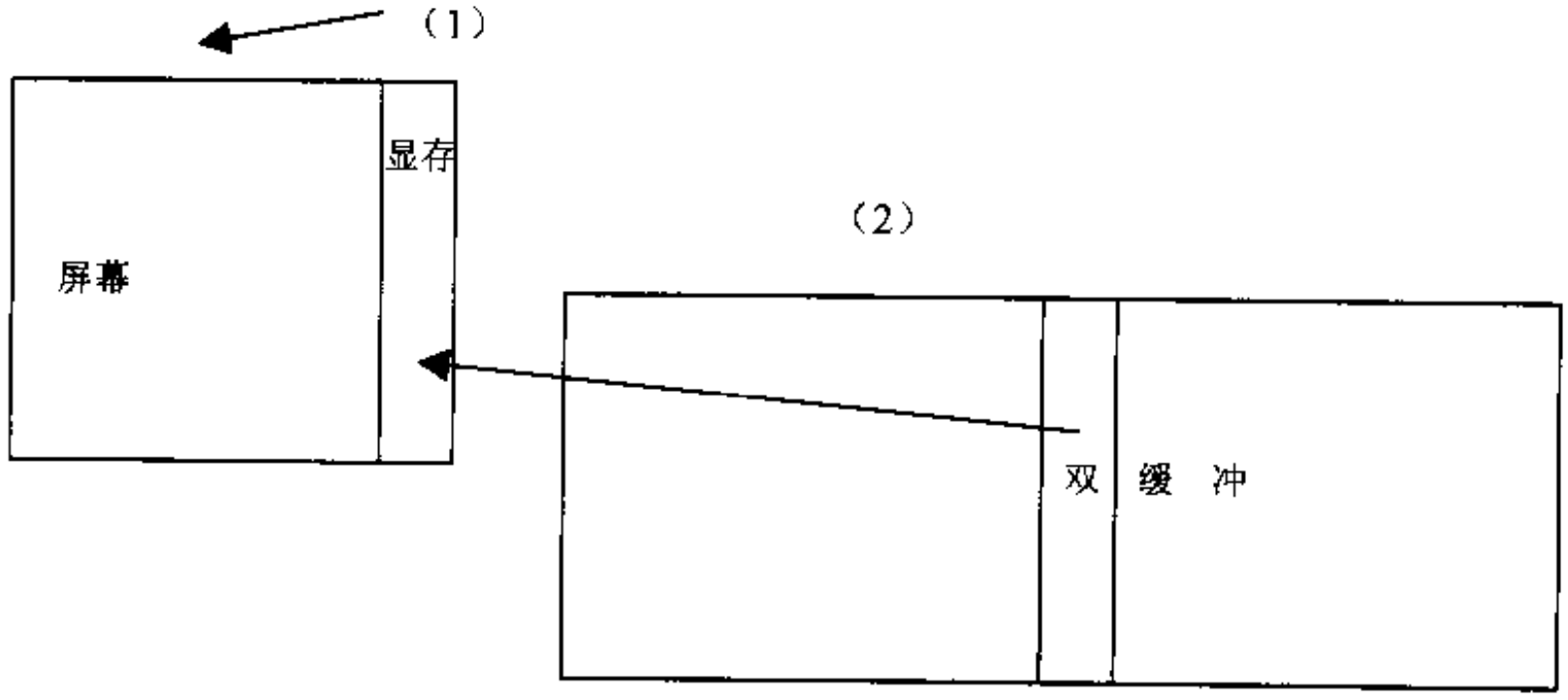


图 9-8　拉屏实现示意

拉屏技术在实际游戏应用中经常被分为 3 种：横向拉屏、纵向拉屏和全方向拉屏。

横向拉屏最早出现在“魂斗罗”游戏中，而后在“MARIO”、“沙罗曼蛇”等游戏中都有使用；纵向拉屏主要出现在一些战斗、比赛类游戏比如“赛车”、“桥砖块”和许多飞行游戏中；全方向拉屏常常被使用在 RPG 游戏的地图大场景中，“仙剑”、“C&C”以及前期的“掘金者”的游戏都属于全方向拉屏。横向拉屏的图例如图 9-9 所示。



图 9-9　横向拉屏

以下是拉屏技术的过程：

（1）在内存中建立大型地图；

（2）将当前地图中当前视区调入显存；

（3）接受用户方向信息；

（4）计算新的地图当前视区；

（5）将原先屏幕上已经有的地图当前视区进行偏移，使其符合新的视区的位置要求；

（6）将屏幕上缺少的地图当前视区内容复制到显存对应位置；

（7）循环 3~6 步骤，直到游戏结束。

横向拉屏和纵向拉屏的实现思路是基本一样的，而全方向拉屏是在这两种拉屏的基础上实现的，可以看作一次横向拉屏和一次纵向拉屏的组合。由于全方向拉屏需要大量的内存空间来存放地图，这往往是常规内存所不能承受的，所以全方向拉屏的例程将在第 13 章内存技术中介绍。

由于拉屏技术是在重画技术的基础上改进而成的，所以所有形式的拉屏技术都可以用重画技术的方法实现。只是重画技术是将整个显示区域按照新的位置重新从内存复制显存，而拉屏技术则仅仅从内存复制视区内没有的新画面内容到显存。两种方法在不同情况下各有自身的优点。这里分别给出重画技术的横向移动和横向拉屏的两个例子，可以比较一下它们的实现方法和效果。

1. 横向重画

横向重画采用每次重新复制当前地图屏幕视区的方法来实现横向滚动效果，以下给出显示地图视图区的函数（针对图像宽度为 640 设计）：

```
void Show_View_Port_Pcx(pcx_picture_ptr background_pcx,int pos) {
unsigned int y,scroll_off,screen_off,flag=0;
if((pos+320)>=SCROLL_WIDTH)
flag=1;
for(y=0;y<SCROLL_HEIGHT;y++) {
scroll_off=((y<<9)+(y<<7)+pos);//设置图像内存列起始地址
screen_off=(((y+150)<<8)+((y+150)<<6));//设置屏幕显存列起始地址
if(flag==0)
_fmemmove((void far *)&video_buffer[screen_off],
(void far *)&background_pcx->buffer[scroll_off],320);//起始位置 0~320 画面显示
else {
_fmemmove((void far *)&video_buffer[screen_off],
(void far *)&background_pcx->buffer[scroll_off],640-pos);//起始位置到 640 画面显示
_fmemmove((void far *)&video_buffer[screen_off+640-pos],
(void far *)&background_pcx->buffer[scroll_off-pos],pos-320);//0 到终止位置显示
} } }
```

此函数通过循环累加计算内存中在视区行首和显存视区行首的地址，而后进行内存向显存的视区行复制来实现当前视区的显示。其中的标志量 flag 用于判断视区是否跨越了图像尾部和图像首部，如果跨越了则要进行两次内存复制工作，分别将视区在图像内当前行首到当前图像行尾和图像当前行首到视区在图像内行尾复制到屏幕。

以下给出一个横向重画的例程：

程序功能：在 320*200 宽度的屏幕下以重画技术实现一张 640*50 的 pcx 文件的横向滚动显示。

程序流程：

（1）初始化存放 pcx 图像的内存空间；

（2）将 pcx 图像调入内存；

（3）显示地图视区；

（4）计算地图视区新的起始点；

（5）循环 3、4 步骤直到整张地图被显示。

主要函数：

```
void PCX_Init_Scroll(pcx_picture_ptr image); //申请长条 pcx 图像内存空间
void PCX_Load_Scroll(char *filename, pcx_picture_ptr image,int enable_palette); //将 pcx 图像保存到内存
void Show_View_Port_Pcx(pcx_picture_ptr background_pcx,int pos);//显示当前视区内的图像
```

程序要点：由于可申请到的常规内存的限制，我们采用了 640*50 的 pcx 文件作为横向拉屏的图像，而视区则是在屏幕下端的 320*50 的区域。

程序代码 move.c 请查阅所附光盘的“source\9”目录。

2. 横向拉屏

横向拉屏和横向重画有所不同。横向重画每次都完全从内存中重新读取下一次要显示于屏幕的所有内容，而横向拉屏由屏幕内自复制和内存补充两部分实现。横向拉屏通过对地图显示区域内部进行类似于拉动的显存复制操作来实现绝大多数视区图像的重显。以下给出显存向左复制函数，复制区域为（1,150）~（319，199）复制给(0,150)~(318,199)：

```
void Move_Screen_Left(int step) {
unsigned int first=320*150,x,y;
for(y=0;y<50*320;y=y+320)
    for(x=0;x<319;x++)
        video_buffer[first+y+x]=video_buffer[first+1+y+x];//显存点左复制
}
```

函数对屏幕最下方的视区基于行的基础上通过点向前复制的方法来实现图像大画面左移一个列。如果移动步长需要变化我们可以通过修改函数来实现。

现在的问题是还有新的一列（第 200 列）需要从图像内存复制进来。新进入视区的极少数内容通过重画方法补充进来，以下给出重画补入图像的函数：

```
void Show_View_Port_Pcx_Left(pcx_picture_ptr background_pcx,int pos) {
unsigned int y,scroll_off_left,screen_off_left,flag=0;
for(y=0;y<SCROLL_HEIGHT;y++) {
scroll_off_left=((y<<9)+(y<<7)+pos);//图像内存中当前行补入点地址
screen_off_left=(((y+150)<<8)+((y+150)<<6)+319);//屏幕中当前行点地址
video_buffer[screen_off_left]=background_pcx->buffer[scroll_off_left];//复制点
} }
```

函数将屏幕视区最右端的一列图像从内存复制到屏幕。当然我们完全可以根据需要修改移动的步长，只是当步长超过 1 的时候将会碰到图像边界处理问题。

以下给出程序：

程序功能：在 320*200 宽度的屏幕下以横向拉屏技术实现一张 640*50 的 pcx 文件的横向滚动显示。

程序流程：

（1）初始化存放 pcx 图像的内存空间；

（2）将 pcx 图像调入内存；

（3）显示地图视区；

（4）计算地图视区新的起始点；

（5）在视区内进行屏幕左移；

（6）对新进入视区的图像进行内存复制；

（7）循环 3、4、5 步骤直到整张地图被显示。

主要函数：

```
void Move_Screen_Left(int step); //在视区内进行屏幕左移
void Show_View_Port_Pcx_Left(pcx_picture_ptr background_pcx,int pos); //对新进入视区的图像进行内存复制
```

程序要点：通过显存与显存的内部复制，大大降低了从内存向显存进行的复制数量。当然同时也增加了一些操作环节。

程序代码 left.c 请查阅所附光盘的“source\9”目录。效果如图 9-10 和图 9-11 所示。

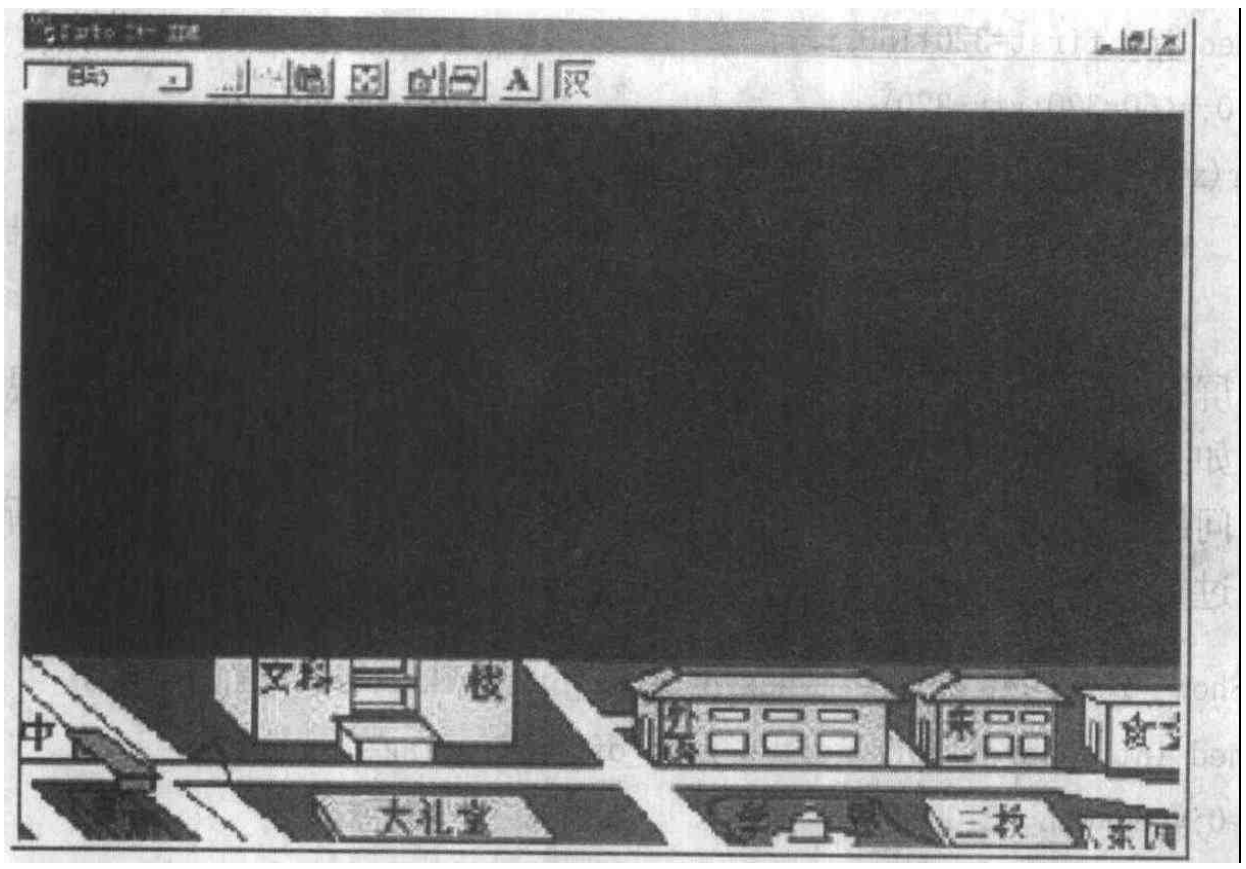


图 9-10　左拉屏（右移）

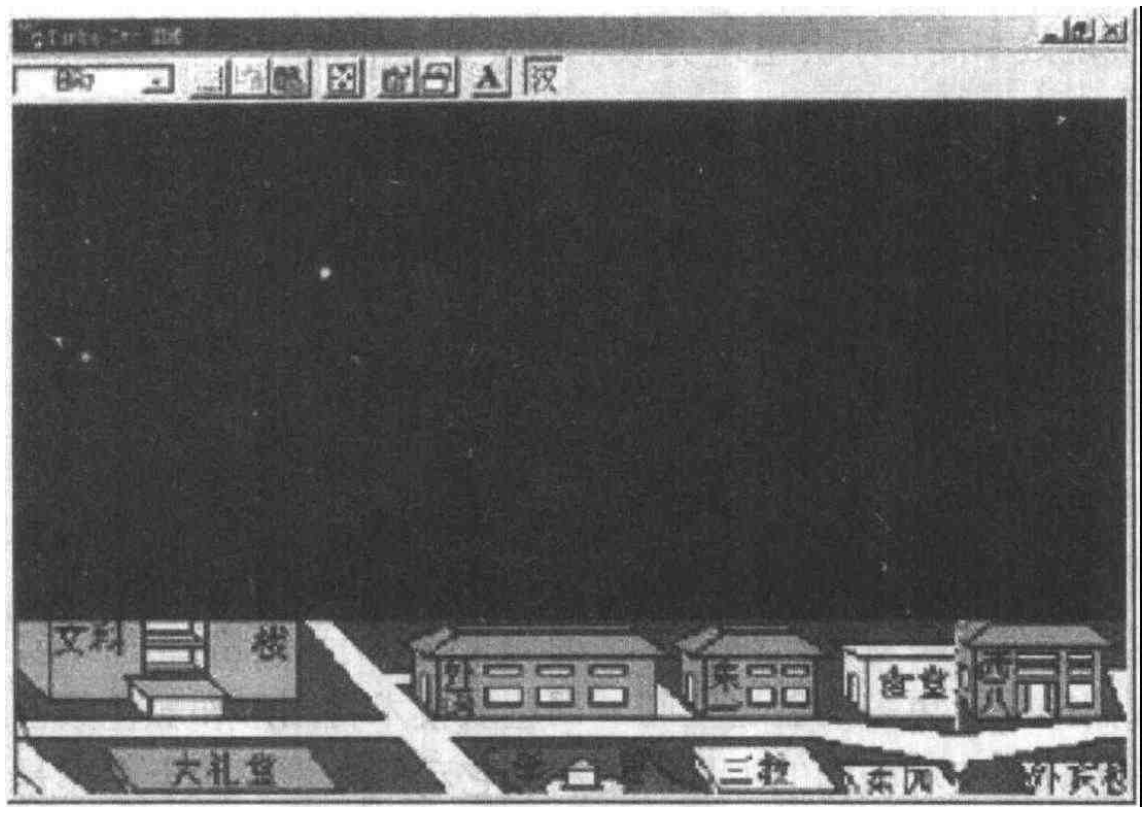


图 9-11　左拉屏（进行中）

9.6　适用环境和效率

为什么要那么多的方法呢？我们发现不同的动画实现方法的适用环境是不同的。重画技术适用于绝大多数的情况，是最常用的动画手段；异或技术较多适用于点和线图形的动画变换，该情况下其效率是很高的；调色板技术较多用于改变颜色不改变图形位置的动画和背景与图形较为简单的移动动画，满足以上条件的情况下它效率最高；拉屏技术只用于整个视区移动的背景画面，这种情况下它的效率通常高于重画技术。

重画技术中直接重画技术很少被采用在真正的游戏中；双缓冲技术是极其常用的一种方法。重画技术的实现手段中几乎只有初学者调用 C 语言图形函数 putimage()来完成动画，它的效率太低了；BIOS 中断直接写屏才是较为专业的选择。

重画技术优点主要在于：

（1）通用性强，任何动画通常都可以用重画思路实现；

（2）实现重画和配合重画技术的办法多种多样。

重画技术的缺点主要在于：

（1）往往有多余绘制内容，事实上在重画的部分总是有可以不要重画的内容；

（2）即便使用了缓冲技术还会造成闪动问题。

解决思路有以下几点：

（1）减小重画区域，用最小的几何形状将需要重画的内容框出，其中一种技术就是下一章要介绍的子画面技术；

（2）对于非常清晰、简练的动画对象可以使用如异或、调色板等其它动画技术来完成；

（3）根据实际情况改变重画的具体操作方法，比如，拉屏技术中的显存内复制代替内存和显存复制就是一个例子。

异或技术对于初学者是一个较为简单、容易掌握和应用的技术，其实现主要是依赖 graphics.h 库函数中的一些函数。

异或技术的优点主要在于：

（1）动画实现效率高，不会出现类似于多余重画的部分；

（2）便于实现点、线和规则图形的动画。

异或技术的缺点主要在于：

（1）图形之间交叉点会出现特殊颜色（异或造成的）；

（2）支持的技术比较少；

（3）对于复杂图形组成的图像和多种颜色组合的图像处理难度较大。

解决思路：

（1）可以自己写各类图形异或函数和异或方式函数；

（2）使用异或技术的图形尽量处于和其他图形较为独立的状态。

调色板技术是我最为喜欢的一种技术。因为它使显示动画的思路最富传奇色彩，同时也给有创意的人们提供了更多的创造空间。调色板技术将图形和颜色紧密地联系起来用以实现动画，使颜色成为实现动画的主导，这和其它动画技术依赖于图形是完全不同的动画实现概念。

调色板技术的优点主要在于：

（1）在动画不出现任何屏幕操作时，一切都早已安排妥当，只需要通过改变调色板就可以实现动画效果；

（2）可以彻底避免闪动的出现。

调色板技术的缺点主要在于：

（1）对于屏幕图形虽然可以改变颜色，但无法改变其预先的图形设置；

（2）对动画编排的设计要求非常高；

（3）只能实现一些特定动画行为。

解决思路：

（1）在调色板技术中加入重画技术中的缓冲思路，将已经制作好的内容交替放入屏幕，从而实现更多的相同位置变换图形动画；

（2）可以引入分层显示的思路，实现多层次动画同时出现。

这里所说的拉屏技术是重画技术衍生出来的。在有些情况下拉屏的方法还可以通过改变屏幕对应显存的位置来实现，这样将更加方便和高效。

拉屏技术的优点主要在于：

（1）将大多数操作放入显存内进行，提高了效率；

（2）适用于达背景动画。

拉屏技术的缺点主要在于：

（1）在计算方面难度要大于重画技术；

（2）操作步骤较重画技术增加了。

解决思路：建立更加通用的各类拉屏函数。

这里还想提一下的就是多种动画技术的共同使用。由于不同动画技术的优缺点不同，所以我们可以根据实际动画情况需要将不同技术同时用于动画中。当然这样做的难度是很大的，其中一些技术所建立的图形不可以与其它技术的图形重合。解决这一问题的最好方法就是分层显示。

分层显示的概念就是把一个二维的屏幕变成一个三维的屏幕，也就是设定多个二维屏幕缓冲并将它们重叠后显示于真正的屏幕上。我们可以对每层虚拟屏幕（其实是内存缓冲）进行一种形式的动画技术操作或者进行一些层面动画对象的绘制，然后根据不同的优先级将它们合并起来。

如果我们要实现街霸中的效果，完全可以将屏幕分成三层。最后一层用调色板技术产生远背景中火光闪动之类的动画效果；第二层用重画技术实现近背景中的一些人物、动物和物体的动画；最前面一层用子画面重画技术实现人物的对打效果。

当然，如果是对于同一种动画技术（或者是类似的动画技术），我们也可以不进行分层，而是使用重画技术中的子画面技术再加上优先级比较来完成。

如果我们要实现 C&C 中的效果，也可以不建立虚拟屏幕，而在屏幕内分层。屏幕最先绘制的是地图画面，而后是坦克子画面和建设的各类基地对象子画面，再是树，最后是飞机，将这些子画面通过优先级计算合并后一起放入屏幕地图之上就可以完成坦克遮住草地；树遮住坦克；飞机遮住树的多维效果了。

9.7　本 章 小 结

计算机游戏中动画的比重非常高，通常一个好的游戏主要看它的创意、美工和动画实现。所以必须考虑使用更多的动画技术来自如地实现我们的创意。C 语言动画技术可以分为：

(1) 重画技术：画了擦，擦了再画，或者一张一张贴上去；

(2) 异或技术：通过对写入点和屏幕点颜色进行逻辑异或运算，来实现擦除和重画运动部分的动画；

(3) 调色板技术：利用预先设置的动作图片和显示适配器中彩色表（以后称调色板寄存器）通过屏幕的颜色变化来实现动画的技术；

(4) 拉屏技术：通过对保存画面的内存和屏幕显存内容的大幅度偏移复制（通常用于大地图的移动）实现的屏幕整体移动效果。

4 种技术中除了调色板技术以外，其他 3 种技术都是重画技术的衍生。事实上重画技术在计算机动画中所占的比例非常高，重画技术还可以继续分：

(1) 直接重画：通过画，清屏，重画，缺点容易产生闪动；

(2) 缓冲：通过开辟一个对应于显存的缓冲内存空间，将所有绘制操作针对缓冲（后台绘制），绘制完成后复制到显存；

(3) 页替换：有的图形模式下显存有两个显存分页（否则可以开辟两个缓冲做弥补），将屏幕首地址交替指向两个显存页（如果是开辟两个缓冲，则是交替复制给显存），并且对当前非显示页进行绘制的动画技术；

(4) 多缓冲：为了使动画画面更加稳定和流畅，在双缓冲技术上开辟更多的对应显存的缓冲空间，建立缓冲顺序依次进行后台同步绘制，从而保证不会出现屏幕等待情况。

各种动画技术都有自身的优点和缺点。作为我本人使用的最早的是异或技术、使用最多的是重画技术，而拉屏技术工作量最大；我最喜欢调色板技术。因为其实现动画的方式非常独特，如果进行完整和精密的策划可以产生意想不到的动画效果。

学后建议

(1) 比较直接写屏技术和重画技术在主流电脑上闪动效果的差距，如果差距不大建议使用该技术作为主要的动画编程技术；

(2) 尝试使用本章提供的异或函数来改写最前面几章的赛车动画和游戏，比较一下两者的闪动情况和程序效率；

(3) 使用调色板技术来制作一个闪动的火焰，当然在此之前你首先需要至少用画笔画一张火焰的双重画面；

(4) 考虑如何制作拉屏地图的程序（后面会介绍使用大内存技术来完成），当然首先需要画一张大地图，然后将它分割成若干个 320*200 像素 256 色的图片。

第 10 章　子画面技术

本章导读

重画技术很少应用于全屏幕重画，更多使用到的是局部重画。这是因为游戏中真正运动的往往只有一些局部对象（比如坦克、人等），而整个屏幕画面背景通常是保持静止的。

在局部重画的实际应用中，除了可以通过二维图形来实现动画效果，更多的时候我们依靠反复调用图形文件中的已经绘制好了的图形对象的各种动作来实现对象运动的效果。这种技术就是本章要介绍的子画面技术。请先看图 10-1。

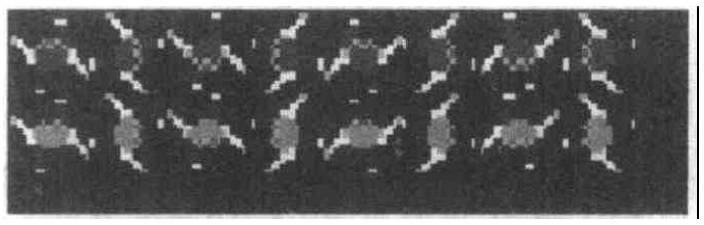


图 10-1　游戏图形对象示意

图片是我用“画笔”绘制的 256 色小人对象，每个小人占 16*16 个像素点。一共两套小人，每套小人有 8 个，每个方向两个，但同一个方向上的动作是不一样的，比如第一行的第一个小人和第五个小人，前者右腿和左手在前，后者左腿和右手在前。如果将两张画面交替在屏幕中进行显示就产生局部动画效果了。

本章就是要介绍如何来实现这种被称为子画面的动画。

本章重点

（1）子画面技术的含义：局部对象重画动画，子画面的实现过程，即调入画面到内存、保存子画面内存、保存背景、绘制子画面导屏幕、延迟、恢复子画面、循环；

（2）子画面技术的几大问题：子画面的显示、运动（键盘控制）以及子画面在地图中的背景处理和在游戏中的计算；

（3）在子画面技术中加入最初步的面向对象思路：对象构建、事件触发、对象行为函数。

10.1　子画面概述

10.1.1　子画面

在上一章动画原理中，我们提到了各类动画实现方法。然而在游戏中应用最广泛的是归类于重画技术的子画面技术。

什么是子画面技术？我们在玩坦克游戏的时候都知道坦克有 4 个运动方向，坦克在不移动的时候也能够产生动画效果。这就是子画面技术的一个应用。

所谓的子画面技术就是让屏幕中某些动画小对象调用图像文件，产生自身的动画和变换规则。具体地说就是将装有一系列子画面的图像文件调入内存数组，然后根据游戏当前进行的情况将其中的某个调入屏幕而产生动画的技术。

之所以使用子画面技术来实现动画的原因也很简单，归结为：

（1）相比全屏重画节约、效率高；

（2）更容易描述动画对象的变化；

（3）有利于成批处理。

这里举一个类似于子画面动画的实际例子：当我们从很高的地方看到马路上一辆汽车向前运动，我们发现马路是相对静止的，而只有很小的那辆汽车在向前移动，移动的过程事实上可以看成汽车不断向前覆盖住其前面的道路，并且将其后面的道路露出来。

那么在屏幕上如何使用子画面来实现这个效果呢？

我们在马路的背景上将汽车子画面放上去，然后改变子画面的位置使它产生向前移动的效果。结果发现：随着汽车子画面的向前移动，子画面的背后留下长长的一条汽车的尾巴，而不是我们所希望的马路。原来我们的屏幕毕竟不是立体的，给汽车子画面遮住的马路部分是无法恢复回来的。

用什么办法来恢复子画面下面的内容？

我们可以在汽车子画面还未画到马路背景的时候，先保存下它将覆盖的区域图像，然后待汽车要移动的时候再将保存下的背景补回整个屏幕，并且再保存下一个子画面背景。这样不断循环就可以完全模仿现实中的运动效果了。

事实上，坦克游戏中坦克就是一个 16*16 像素左右的子画面，而为了实现它的动画我们需要绘制一个拥有至少 8 个子画面的图像文件。这 8 个子画面所不同的是它们一共有上下左右四个方向，其中每两个是朝着一个方向的，而同一方向的两个小图标也有着细微的不同。这样我们就可以通过交替换图标来实现改变方向、前进等动画效果。

那么子画面在整个画面中到底是如何发挥作用产生动画的呢？以下给出一个子画面对象的动画过程（图片示意见图 10-2、10-3 及图 10-4）：

（1）画整个屏幕画面；

（2）从文件保存若干子画面到内存；

（3）保存将要放入子画面位置的背景图像块到内存；

（4）将子画面放入屏幕画面中；

（5）接收游戏者信息或按照规则进行变量计算；

（6）将内存中保存的子画面背景图像块恢复到屏幕；

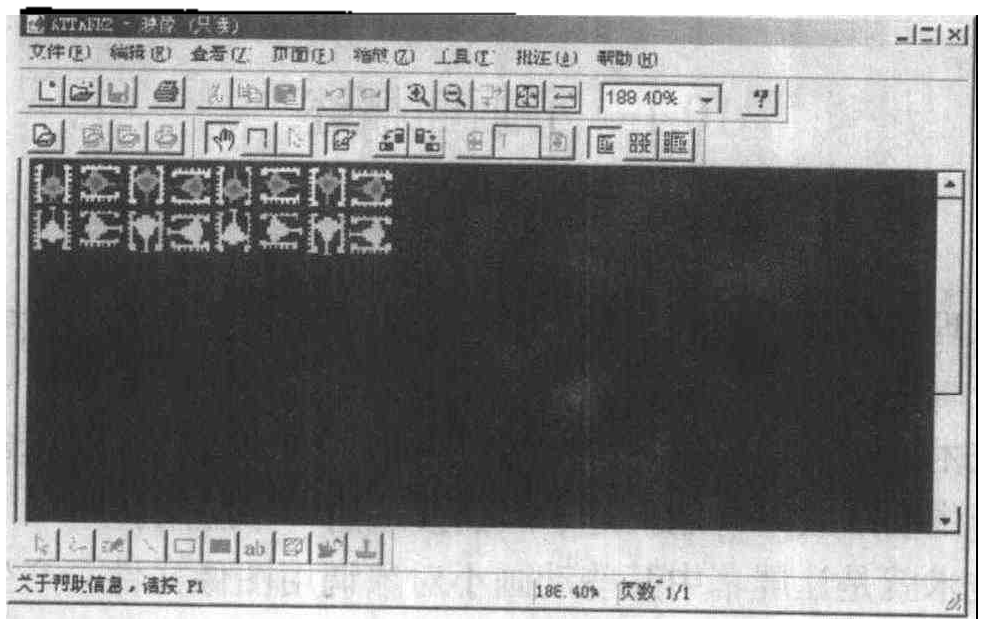


图 10-2　绘制子画面图像

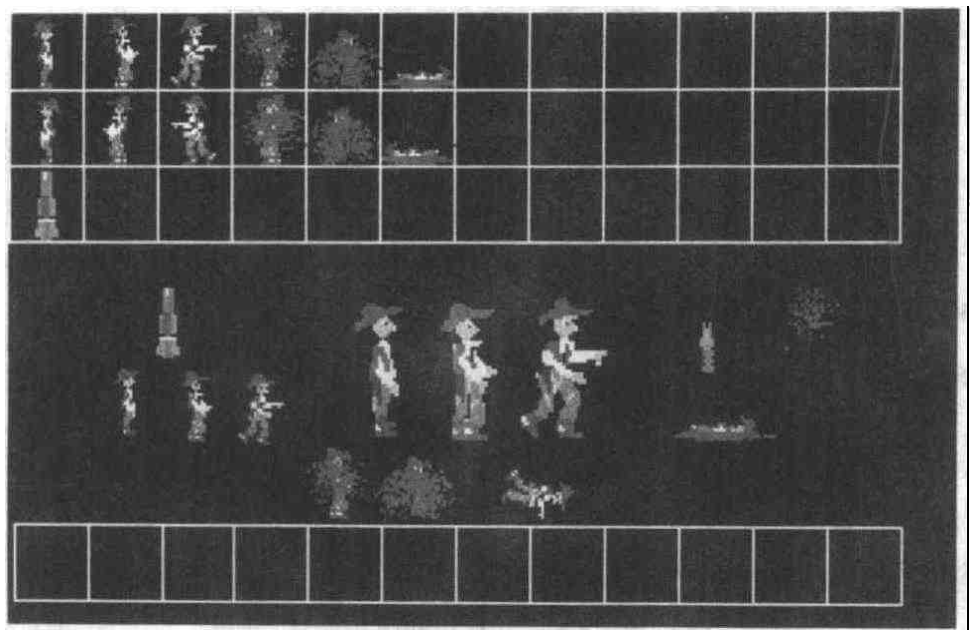


图 10-3　实现动画

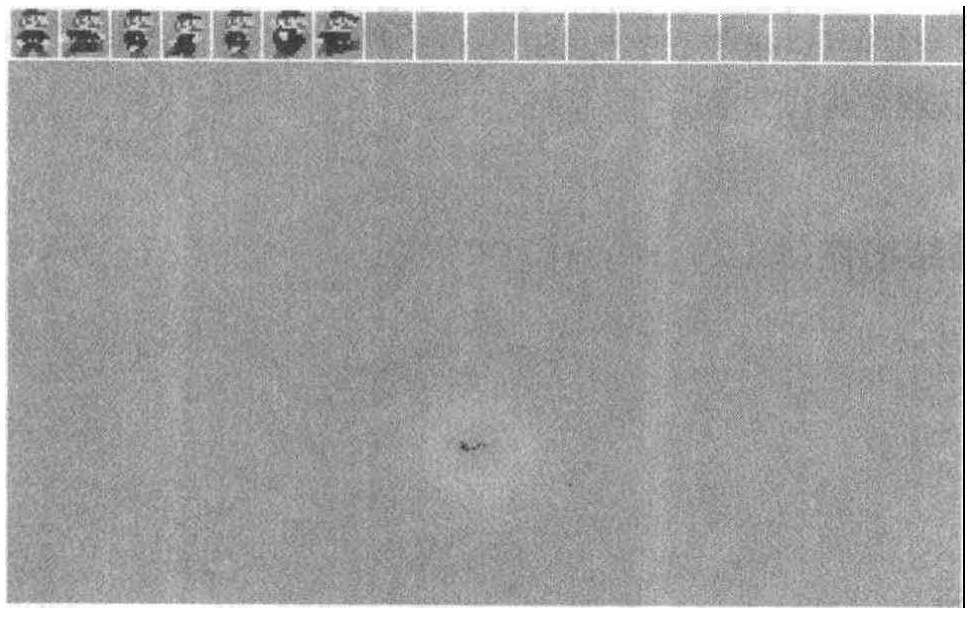


图 10-4　动画模拟示意

（7）处理游戏中的计算问题，并且根据游戏情况确定子画面放入的下一个新位置；

（8）循环 3~7 步骤，直到游戏结束。

10.1.2　子画面结构

子画面实际上就是一种对象，每种对象都有自己的属性和行为。所以无论是坦克还是小人这样的子画面都有共同的一些特征。现在我们的任务就是将子画面的特点抽取出来，写成一个结构体。

我们很容易想到子画面有大小、前次位置、当前位置、宽度高度、点阵画面数量、当前帧号和存放点阵画面的数组以及当前状态等属性。如果再仔细分析一下，还需要保存背景画面、单位触发时钟和移动速度等属性。此外，我们还需要将所有子画面对象都用链表连起来和要一个子画面对象的行为函数。

以下给出子画面通用结构定义：

```
typedef struct sprite_type {
int x,y;//当前位置
int x_old,y_old;//先前位置
int width,height;//宽度和高度
```

```
int anim_clock;//子画面时钟，不常用
int anim_speed;//子画面速度，不常用
int motion_clock;//运动时钟，不常用
int motion_speed;//运动速度，不常用
char far *frames[MAX_SPRITE_FRAMES];//存放子画面点阵的数组指针
int curr_frame;//当前子画面帧号
int num_frames;//子画面总帧数
int state;//当前状态
char far *background;//存放子画面背景的指针
void far *extra_data;//指向特别的数据，不常用
struct sprite_type *next;//链表指针
void (far *spr)(struct sprite_type *spr);//子画面对象行为函数指针
key key;//功能按键结构体变量
}sprite,*sprite_ptr;
```

以下是功能按键结构体：

```
typedef struct key {
char up;//上
char down;//下
char left;//左
char right;//右
char stop;//停止
char quit;//退出
char fight;//战斗或者发射子弹
}key,key_ptr;
```

这样，一个具有一定面向对象特点的子画面结构体就建立起来了。值得一提的是，链表指针和函数指针的使用将使子画面游戏程序从原先的一些子画面程序偏重过程化向面向对象化大大的迈进。这将在稍后详细介绍。

这里还想来介绍一个子画面的特例——子弹。我们玩得最多的便是枪战游戏。枪战中的坦克或者人都可以使用子画面结构体来实现，那么更小的子弹如何描述呢？

当然我们也可以使用上面的子画面结构来描述一颗子弹，但是是否太浪费了一些。我们知道许多子弹仅仅只占用一个像素，而且也没有那么多的动画效果。所以这里给出子弹的子画面点结构：

```
typedef struct direct_type {
int x; //x 可以选择-1,0,1
int y; //y 可以选择-1,0,1
} direct; //子弹的方向
typedef struct bullet_type {
```

```
int x; //子弹当前 x 坐标
int y; //子弹当前 y 坐标
char color; //子弹颜色
char color_gd; //子弹点背后颜色
direct direction; //子弹方向
int flag; //子弹是否激活
}bullet,*bullet_ptr;
```

为了在实际程序中方便使用，我们将子弹方向分为 x,y 两个单位分量来描述，例如(1,0)、(0,1)、(-1,0)、(0,-1)分别表示向右、上、左、下移动，当然需要的时候也可以增加(1,1)这样的 45 度方向。此外，由于子弹只占用一个点，所以其背景的定义其实就是一个点的颜色，这比子画面背景的定义方便了很多。

对子弹的子画面结构定义，我们只要增加一个子弹的结构体变量就可以了：

```
typedef struct sprite_type {
int x,y;//子画面当前位置
int x_old,y_old;//子画面原先位置
int width,height;//子画面宽度和高度
int anim_clock;
int anim_speed;
int motion_clock;
int motion_speed;
char far *frames[MAX_SPRITE_FRAMES];//指向子画面图像内存指针数组
int curr_frame;//当前子画面
int num_frames;//子画面数量
int state;//子画面状态
char far *background;//指向子画面背景内存指针
void far *extra_data;
struct sprite_type *next;//指向下一个子画面对象结构
void (far *spr)(struct sprite_type *spr);//指向子画面对象事件函数指针
key key;//使用到的游戏控制按键
bullet bullet;//增加的子弹结构体变量
}sprite,*sprite_ptr;
sprite_ptr head,now,pre;
```

10.1.3　面向对象

程序设计的两大主流事实是，过程化和面向对象。最初所有的程序都是过程化的，这也就是为什么叫“程序”的原因。之后人们逐渐发现面向对象的思路更加有利于设计真正大型实用的程序，伴随着 Windows 的出现，面向对象语言基本取代了过程化语言。

然而如同在对象化语言中有处理过程，子过程化语言中也可以应用到对象的思路。事

实上 C 语言中的结构体已经是对象概念的一个雏形了，只是它还没有包含自己的函数。不过不要紧我们可以使用指向函数的指针来解决这个问题。于是在 C 语言中更多地使用面向对象的思路是完全可行的。

本章在介绍子画面技术中力求最大突破就是将面向对象的思路贯穿到子画面技术中。由于 C 语言毕竟是过程化语言，所以我们的所谓的面向子画面对象实际上也是在过程的基础上融入一些面向对象化的思路。

还是从整个子画面游戏的过程来谈这个问题。相比从前单纯从过程化角度设计的子画面程序，这里在若干问题上使用到了面向对象概念：

（1）子画面的定义完善对象化。

在对子画面的定义过程中，除了定义了子画面尽可能多的属性以外，还增加了一个指向子画面事件函数的指针，对应子画面功能按键序列和一个子画面对象链表。

（2）在子画面建立过程中强调面向对象概念。

在子画面构建中，首先为具体对象的结构体变量申请空间，并且完成所有如同子画面内存设置等工作、链表设置工作和已经准备好了子画面事件函数。

（3）在游戏主程序中通过检测函数循环调用链表来检验每个子画面动画。

只是在这里我们将具体键盘事件触发放到子画面事件函数中去了，而不是放在检测函数中（根据检测去触发对应子画面事件）。这是由于 C 语言在事件触发上的一些限制和子动画程序实际调试效果以及子画面对象在每次循环中都有事件发生等原因决定的。检测函数中事件触发的例子可以在稍后的第 15 章界面技术中看到，因为鼠标触发事件和界面对象的特点比较适合事件触发。

（4）为每个子画面都设计了事件函数。

不将子画面的行为放在游戏主循环中作为过程处理。而是通过主循环中的检测函数进行依次调用。

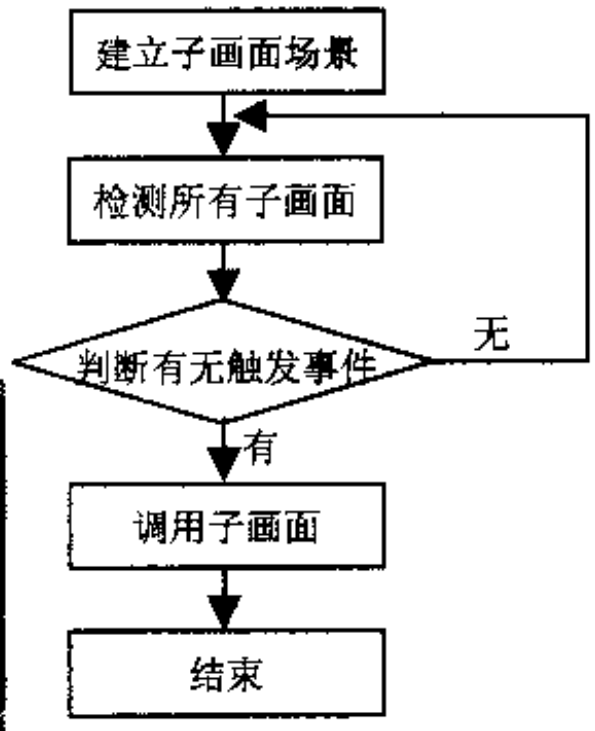


图 10-5　建立子画面流程示意

（5）在子画面对象死亡（释放）时，释放对象所有空间。

同时，还将对象从链表中剔除。

这里值得一提的是，由于子画面在每次检测循环中都会有事件动作触发的特点，我们没有采用事件触发的方法。而是将检测事件写入子画面事件函数中，并判断具体触发事件。

以上几点的改进大大增强了游戏的清晰程度，在游戏的维护和扩充方面也非常方便。

以下从面向对象的角度来看整个子画面程序的结构，如图 10-5。

10.2　显示子画面

大家也许觉得子画面技术似乎要做的事情非常多，又要保存背景、又要考虑游戏中子弹、碰撞等问题，真的很难下手。这里将暂时摒弃这些复杂的问题而只想办法显示出一个子画面来。

显示子画面的步骤非常简单：

（1）初始化 320*40 的 pcx 图片内存空间；

（2）将 320*40 的 pcx 文件（8 个子画面都在这个文件中）调入该内存空间；

（3）初始化子画面空间（8 个子画面空间）；

（4）将内存中的图片按照 16*16 的大小对应复制给子画面空间；

（5）删除为 pcx 图片申请的内存空间；

（6）按键则顺序显示 8 个子画面图片；

（7）删除所有子画面空间。

关于初始化 pcx 文件内存、将 pcx 文件调入内存和删除为 pcx 图片申请的内存等问题我们都在第 8 章介绍 pcx 文件的时讲过了。这里之所以调用 320*40 的 pcx 图片也是因为内存无法一次申请到 320*200 的空间。

1. 初始化子画面空间

初始化子画面空间实际上就是给子画面结构体变量赋初始值，并且申请必需的空间。其函数如下：

```
void Sprite_Init(sprite_ptr sprite,int x,int y,int ac,int as,int mc,int ms,void (far
*spr),char key[7]) {
int index;
sprite->x= x;//初始化子画面横坐标
sprite->y= y; //初始化子画面纵坐标
sprite->x_old= x; //初始化子画面原先的横坐标
sprite->y_old= y; //初始化子画面原先的纵坐标
sprite->width= SPRITE_WIDTH; //初始化子画面宽度
sprite->height   = SPRITE_HEIGHT; //初始化子画面高度
sprite->anim_clock   = ac;
sprite->anim_speed   = as;
sprite->motion_clock = mc;
sprite->motion_speed = ms;
sprite->curr_frame   = 0; //初始化当前子画面
sprite->state= SPRITE_DEAD; //初始化子画面状态
sprite->num_frames = 0; //初始化子画面数量
sprite->background = (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT+1);//初始化子画面
sprite->spr=spr; //初始化子画面事件函数
```

```
sprite->key.up=key[0];//初始化子画面向上控制键
sprite->key.down=key[1]; //初始化子画面向下控制键
sprite->key.left=key[2]; //初始化子画面向左控制键
sprite->key.right=key[3]; //初始化子画面向右控制键
sprite->key.stop=key[4]; //初始化子画面停止控制键
sprite->key.quit=key[5]; //初始游戏退出控制键
sprite->key.fight=key[6]; //初始化子画面射击控制键

for (index=0; index<MAX_SPRITE_FRAMES; index++)////初始化子画面图像指针指向空
sprite->frames[index] = NULL;
}
```

建立一个子画面空间的代码段：

```
char key[7]={'8','2','4','6','5','q','0'}; //设置功能按键
PCX_Load_Screen("attank2.pcx",1); //读取画有 8 个子画面的 pcx 文件
now=pre=(struct sprite *)malloc(sizeof(struct sprite_type)); //为子画面结构体变量申请
空间
head=now; //当前子画面对象为链表头
Sprite_Init(now,0,0,0,0,0,0,spr,key); //初始化子画面结构体变量
now->next=NULL; //由于只申请一个子画面对象，链表下一个指向空
```

2. 从图像内存读入子画面内存

由于 pcx 文件已经被读入内存，现在要做的事情就是将整个图片切割成一个一个的子画面读入为子画面申请的若干内存中。

事实上我们在图中可以看出 8 个子画面被绘制在 320*200 的 pcx 文件的左上角。以 8 个 16*16 子画面为例，它们的起始位置分别是(0,0)、(16,0)、(32,0)、(48,0)...

我们的读取方法也就出来了：

（1）为当前子画面申请子画面宽度乘以子画面高度的空间（比如 16*16）；

（2）根据当前子画面帧号计算出其在图像中的起始位置（比如帧号为 3，起始位置为(32,0)；

（3）从 pcx 内存计算出的偏移（帧号 3 偏移 32）开始读取 16 个字节到子画面内存；

（4）pcx 内存从刚才计算出的偏移位置再偏移 320（屏幕宽度）；

（5）从 pcx 内存当前偏移再读取 16 字节到子画面内存当前偏移；

（6）循环 4、5 步骤直到读完所有子画面行（比如 16 行）。

以下给出具体函数：

```
void PCX_Grab_Bitmap(pcx_picture_ptr image, sprite_ptr sprite, int sprite_frame,int
grab_x, int grab_y) {
int x_off,y_off, x,y, index;
char far *sprite_data;
```

```
//申请子画面空间
sprite->frames[sprite_frame] = (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT);
sprite_data = sprite->frames[sprite_frame];
//确定子画面在图像内存中的偏移
x_off = (SPRITE_WIDTH) * grab_x;
y_off = (SPRITE_HEIGHT) * grab_y ;
y_off = y_off * 320;
//将图像内存中的对应部分读入子画面内存中
for (y=0; y<SPRITE_HEIGHT; y++) {
for (x=0; x<SPRITE_WIDTH; x++) {
   sprite_data[y*SPRITE_WIDTH + x] = image->buffer[y_off + x_off + x];
   } y_off+=320;
} sprite->num_frames++;//子画面数增加 1
}
```

前面我们提到内存无法申请 320*200 空间，于是我们采用了变通的方法申请了 320*40 的空间。有些人干脆不申请空间，而直接将 pcx 文件读取到屏幕显存，然后再从屏幕显存读取子画面到子画面内存。这时，我们只需要将上面的函数修改一下，也就是把指向 pcx 内存的指针指向显存就可以了。直接从屏幕读取子画面的函数如下：

```
void PCX_Grab_Bitmap_Screen(sprite_ptr sprite,int sprite_frame,int grab_x, int grab_y){
int x_off,y_off, x,y, index;
char far *sprite_data;
sprite->frames[sprite_frame] = (char far *)malloc(SPRITE_WIDTH * SPRITE_HEIGHT);
sprite_data = sprite->frames[sprite_frame];
x_off = (SPRITE_WIDTH) * grab_x;
y_off = (SPRITE_HEIGHT) * grab_y ;
y_off = y_off * 320;
for (y=0; y<SPRITE_HEIGHT; y++) {
for (x=0; x<SPRITE_WIDTH; x++) {
//从屏幕读取图像到子画面内存
   sprite_data[y*SPRITE_WIDTH + x] = video_buffer[y_off + x_off + x];
   } y_off+=320;
} sprite->num_frames++;
}
```

连续直接从屏幕读取所有子画面帧到内存的代码段如下：

```
for(index=0;index<MAX_SPRITE_FRAMES;index++) { //从第一个开始读取所有帧
 PCX_Grab_Bitmap_Screen(now, index, index, 0); //从屏幕读取当前帧到内存
}
```

3. 显示子画面

显示子画面实际上就是将当前帧号子画面从对应序号的子画面内存复制到屏幕要求的位置，这与前面的从屏幕显存读取子画面到子画面内存的过程正好相反。这里就不做赘述了。给出函数如下：

```
void Draw_Sprite(sprite_ptr sprite) {
char far *work_sprite;
int work_offset=0,offset,x,y;
work_sprite = sprite->frames[sprite->curr_frame];
//计算子画面在显存中的偏移
offset = ((sprite->y) << 8) + ((sprite->y) << 6) + sprite->x;
//显示子画面到屏幕
for (y=0; y<SPRITE_HEIGHT; y++) {
for (x=0; x<SPRITE_WIDTH; x++) {
  if(work_sprite[work_offset+x])//判断子画面当前点是否是黑色
//如果不是黑色则显示该点，如果是黑色则保留屏幕点（就是透明的意思）
 video_buffer[offset+x]=work_sprite[work_offset+x]; }
//屏幕地址和子画面地址进行行偏移计算
offset  += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} }
```

4. 删除子画面内存

删除子画面其实就是释放子画面对象调用中申请的包括背景、所有子画面帧和子画面结构体变量等的内存：

```
void Sprite_Delete(sprite_ptr sprite) {
int index;
farfree(sprite->background);//释放子画面背景空间
//释放所有子画面图像空间
for (index=0; index<MAX_SPRITE_FRAMES; index++)
farfree(sprite->frames[index]);
free(sprite);//释放子画面对象结构
}
```

以下给出一个子画面显示（从显存读取到子画面内存）的程序：

程序功能：通过调用 pcx 图像文件（见图 10-6）建立一个 8 张图片的子画面系列，并将所有子画面顺序显示于屏幕。

程序流程：

（1）将 320*200 的 pcx 文件（8 个子画面都在这个文件中，如图 10-6）调入显存；

（2）初始化子画面空间（8 个子画面空间）；

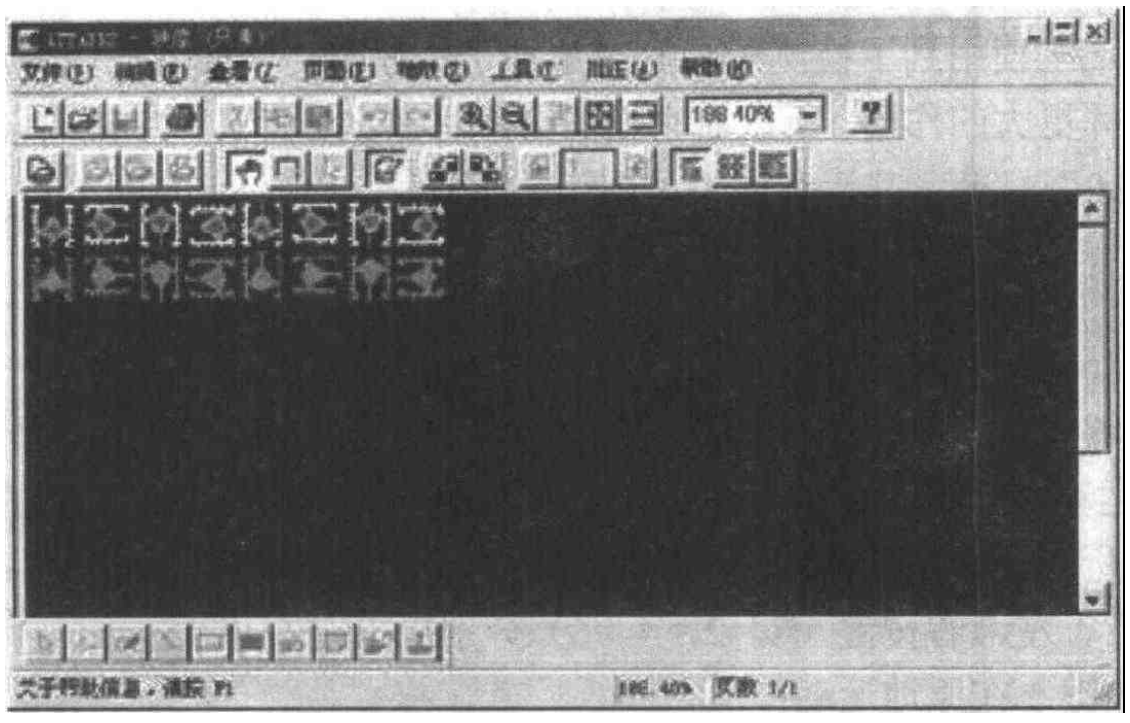


图 10-6　pcx 图像文件

（3）将内存中的图片按照 16*16 的大小对应复制给子画面空间；

（4）按键则顺序显示 8 个子画面图片；

（5）删除所有子画面空间。

主要函数：

```
void PCX_Load_Screen(char *filename,int enable_palette); //将 pcx 文件存入内存
void Sprite_Init(sprite_ptr sprite,int x,int y,int ac,int as,int mc,int ms); //初始化子画面空间
void PCX_Grab_Bitmap_Screen(sprite_ptr sprite, int sprite_frame,int grab_x, int grab_y);
 //从显存读取子画面到子画面空间
void Draw_Sprite(sprite_ptr sprite); //绘制子画面图片
void Sprite_Delete(sprite_ptr sprite); //释放子画面空间
void Fill_Screen(int value); //填充屏幕
void Judge_Sprite(void); //检测所有子画面对象
void spr(sprite_ptr sprite); //子画面事件
```

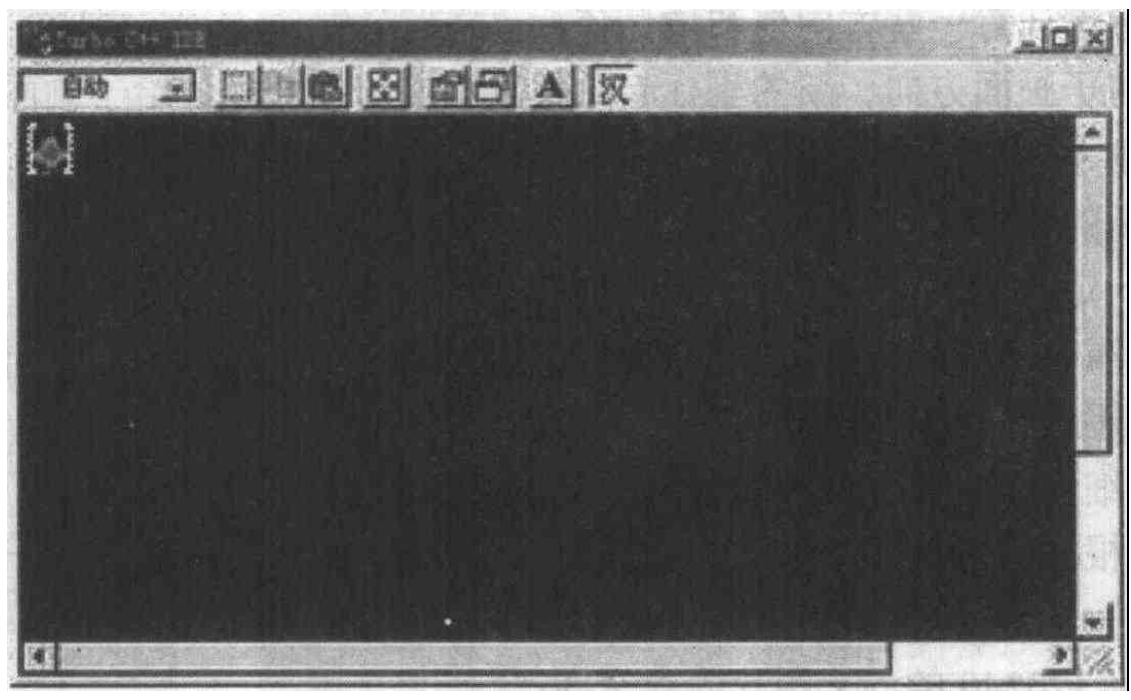


图 10-7　调用子画面的 pcx 图像

程序要点：这里使用到了检测子画面对象和子画面事件函数，它将在稍后的多个子画面对象程序中发挥更大的作用。所以对它们的讲解将放到后面。程序代码 son.c 请查阅所附光盘的“source\10”目录。其运行效果如图 10-7 所示。

10.3　子画面运动

我们已经能够显示所有子画面了。现在的任务是要如同坦克游戏一样让子画面运动起来。要让子画面运动无非就是根据用户按键对子画面的起始坐标和子画面的帧号进行计算。

首先我们将所有子画面接收用户键盘输入面产生的运动情况进行罗列：

（1）上、下、左、右按键（8，2，4，6）分别上下左右移动；

（2）停止按键 5 在原地运动；

（3）无按键则保持原先运动状态；

（4）过多按键则忽略；

（5）退出按键 q 则终止游戏。

由于在定义子画面结构的时候考虑了游戏功能按键问题，所以设置功能按键只需要建立一个键值数组就可以了。具体设置功能键的代码段如下：

```
char key[7]={'8','2','4','6','5','q','0'}; //定义 7 个功能键
...
Sprite_Init(now,0,0,0,0,0,0,spr,key); //初始化子画面结构
```

在稍候的子画面事件函数中将非常容易地根据用户输入按键和子画面功能按键的比较来得出子画面的具体事件运动结果。

如同上一个例程，我们将子画面运动行为的代码写在对应子画面的事件函数中（上一个程序子画面事件函数是 spr()）。由于 spr()非常简单，只要在每次按键情况下显示下一帧子画面图片就可以了。而在本程序中我们才真正涉及子画面的一些事件行为，具体处理过程如下：

（1）判断是否有按键，无则到第 3 步；

（2）根据按键计算一些方向值和增量值；

（3）将当前子画面方向、新的坐标起始位置和当前帧号赋值给子画面结构变量；

（4）删除刚才的子画面屏幕；

（5）显示新的子画面；

（6）延迟一定时间。

由于子画面事件函数主要涉及对用户按键判断和坐标增量与方向变量的基本计算，并且函数较长（但不难），所以不在这里个别列出代码分析了。

以下给出详细的子画面接收用户输入运动的程序：

程序功能：根据键盘按键（四个方向按键和一个原地踏步按键）控制子画面的运动。

程序流程：

（1）初始化 320*40 的 pcx 图片内存空间；

（2）将 320*40 的 pcx 文件调入该内存空间；

（3）初始化子画面空间；

（4）将内存中的图片按照 16*16 的大小对应复制给子画面空间；

（5）删除为 pcx 图片申请的内存空间；

（6）判断是否有按键，如果没有则调用刚才方向上的另一子画面，并且返回 6；

（7）如果有按键则，4 表示向左，6 表示向右，8 表示向上，2 表示向下，5 表示原地，q 表示退出程序；

（8）任意一个方向运动处理如下：

a. 标示当前子画面方向；

b. 计算相应的移动位置；

c. 计算当前应该调用的子画面图片序号（一个方向也有 2 个子画面）；

（9）根据新的位置、当前子画面序号绘制子画面到屏幕对应位置；

（10）循环 6~9 步骤直到退出按键被触发；

（11）删除所有子画面空间。

主要函数：

```
void PCX_Init(pcx_picture_ptr image); //初始化 pcx 文件内存空间
void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette);
//从 pcx 图像内存读取子画面到子画面内存
void PCX_Delete(pcx_picture_ptr image); //删除为 pcx 申请的内存
void Clear_Key_Buffer(void); //清除键盘缓冲（将在第 14 章详细介绍）
```

程序要点：

（1）在某一个方向上的运动或者原地踏步，必须交替显示该方向上的两个略有差别的子画面图像方能产生子画面在不断运动的效果。这可以通过对子画面序号加模运算（模为 8）或者在循环中置奇偶标志位来实现。

（2）由于要将 pcx 图片内事先画好的子画面图像放入子画面空间首先需要将整个文件放入内存空间。但是 320*200 的 pcx 图片对应的常规空间是申请不到的，所以把图片都设置为 320*40，以便申请到对应大小的内存空间。这里把使图片高度改成 40，原因是这里使用到的子画面是 16*16 点阵的，所以即使两排其高度也只有 32。

（3）在变量每次接收完键盘按键后清除键盘缓冲区可以防止连续按键造成的程序停顿问题。

（4）在 spr()函数中在给子画面赋值前还要检测子画面是否超出 320*200 屏幕，如果超出则保持原来坐标。

程序代码 sonkey.c 请查阅所附光盘的“source\10”目录。

10.4　背 景 问 题

在第 10.1 节我们已经提到过解决子画面运动时背景无法恢复的问题及解决方法，即我们可以在子画面写入屏幕前保存它要覆盖位置的背景块到内存，然后等待子画面移动时先将内存中的背景块贴回原先位置（此时的屏幕上面没有子画面了），接着保存子画面新位置

的背景块，最后在新位置显示子画面。这样循环往复将解决背景无法恢复问题。

从程序角度上来说，解决背景问题的这个方法仍然是写在子画面事件函数中（例程中的 spr()函数），因为每个子画面的所有变化行为都在它的事件函数中完成。这里我们希望在此基础上详细地来讨论这个子画面事件函数。以下是一个较为完整的子画面事件函数处理过程：

（1）恢复子画面后的背景块（第一个背景块在主程序开始时就进行保存了）；

（2）判断是否有按键，无则到第 3 步；

（3）根据按键计算一些方向值和增量值；

（4）将当前子画面方向、新的坐标起始位置和当前帧号赋值给子画面结构变量；

（5）保存子画面新位置上的背景；

（6）显示新位置上的子画面；

（7）延迟一定时间。

将这个过程和上一节子画面运动中的子画面事件过程相比较，我们发现这里只是增加了恢复子画面和保存子画面的部分。之所以在一开始就恢复子画面背景块的原因是，此时的子画面坐标还是老位置的，而不是在计算后的新位置。此外保存子画面背景的任务却一定要在完成子画面新位置计算之后进行，否则保存的位置还是老位置，这样将毫无意义。

在这个过程中将涉及到两个新的子画面函数：保存和恢复背景块函数。这两个函数非常类似于将 pcx 图像存入子画面内存函数和绘制子画面函数。这里将只给出函数代码而不进行一一介绍了。

保存背景块函数：

```
void Behind_Sprite(sprite_ptr sprite) {
char far *work_back;
int work_offset=0,offset,y;
work_back = sprite->background;
//计算子画面在屏幕中当前偏移
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
//将屏幕即将被子画面遮盖的画面保存到背景内存
for (y=0; y<SPRITE_HEIGHT; y++) {
//复制一行屏幕子画面背景到背景内存
_fmemmove((void far *)&work_back[work_offset],(void far *)&video_buffer[offset],
SPRITE_WIDTH);
//行偏移计算
offset  += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} }
```

恢复背景块函数：

```
void Erase_Sprite(sprite_ptr sprite) {
```

```
char far *work_back;
int work_offset=0,offset,y;
work_back = sprite->background;
offset = (sprite->y << 8) + (sprite->y << 6) + sprite->x;
//将背景内存恢复到屏幕子画面上（如同补草皮一样）
for (y=0; y<SPRITE_HEIGHT; y++) {
//复制一行背景内存到屏幕子画面位置
_fmemmove((void far *)&video_buffer[offset],
     (void far *)&work_back[work_offset],
     SPRITE_WIDTH);
//计算行偏移
offset  += SCREEN_WIDTH;
work_offset += SPRITE_WIDTH;
} }
```

以下给出带背景的子画面程序：

程序功能：在键盘控制子画面运动的基础上，保存并恢复在运动中被子画面遮挡了的屏幕背景画面。

程序流程：

（1）初始化 320*40 的 pcx 图片内存空间；

（2）将 320*40 的 pcx 文件调入该内存空间；

（3）初始化子画面空间；

（4）将内存中的图片按照 16*16 的大小对应复制给子画面空间；

（5）删除为 pcx 图片申请的内存空间；

（6）保存当前子画面位置背景画面到内存；

（7）恢复刚才保存的子画面背景画面；

（8）判断是否有按键，如果没有则设置当前方向上的另一个子画面为当前子画面，并转到 11；

（9）如果有按键则，4 表示向左，6 表示向右，8 表示向上，2 表示向下，5 表示原地，q 表示退出程序；

（10）任意一个方向运动处理如下：

a. 标示当前子画面方向；

b. 计算相应的移动位置；

c. 计算当前应该调用的子画面图片序号（一个方向也有 2 个子画面）；

（11）保存新计算出的子画面位置背景画面到内存；

（12）根据新的位置、当前子画面序号绘制子画面到屏幕对应位置；

（13）循环步骤 7~12 直到退出按键被触发；

（14）删除所有子画面空间。

主要函数：

```
void Behind_Sprite(sprite_ptr sprite); //保存子画面后的屏幕画面到内存
void Erase_Sprite(sprite_ptr sprite); //恢复刚才保存的子画面背景到屏幕
```

程序要点：

（1）在这里子画面动画过程较上一个程序复杂了很多，主要由 4 个步骤的循环调用组合实现：

a. 恢复旧背景；

b. 计算新位置和子画面序号；

c. 保存新背景；

d. 显示新子画面。

（2）在主程序子画面构建完成之后，就要保存子画面的第一个背景块，然后再开始游戏主循环。

程序代码 sonback.c 请查阅所附光盘的“source\10”目录。其效果如图 10-8 所示。

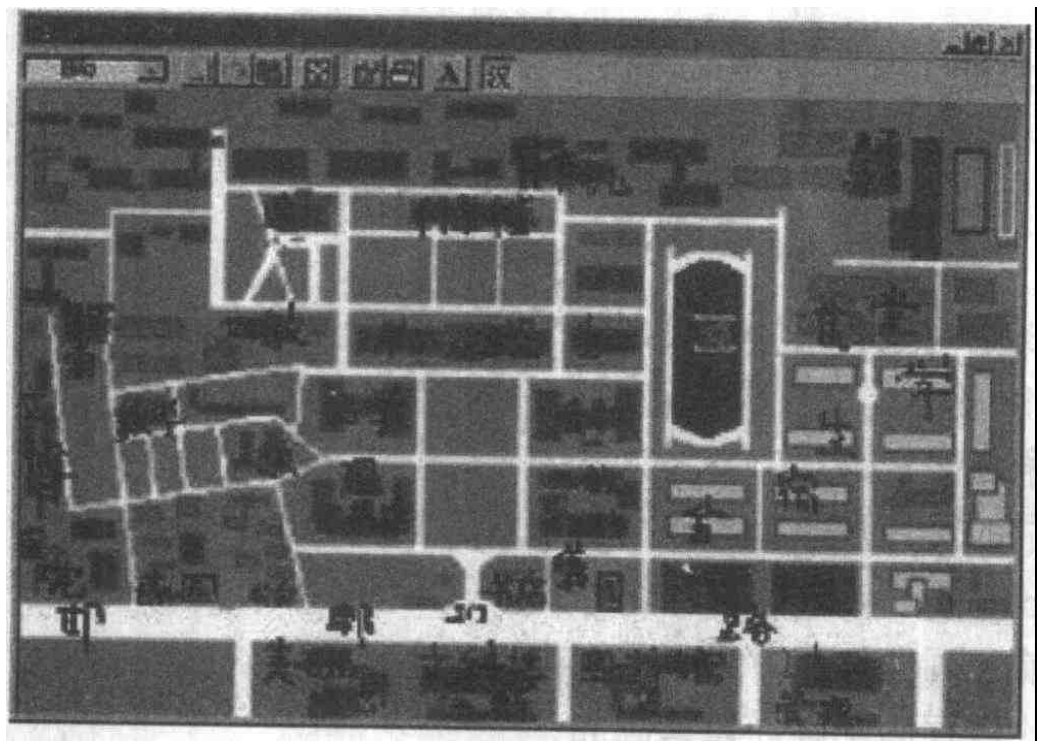


图 10-8　保存并恢复子画面背景

10.5　子画面游戏

我们已经解决了子画面的显示、运动和背景恢复问题，现在可以尝试来做一个子画面的游戏。事实上在本书的最初我们已经提到过动画和游戏只有用户输入这一步之遥，而我们在解决子画面运动问题的时候其实已经解决了用户键盘输入的问题了。现在只需要给游戏增加一些规则和加入一些有趣的东西就可以了。

最后我决定做一个最简单的坦克游戏，它有以下一些特点和规则：

（1）有两辆坦克，其中一辆是用户控制，一辆是计算机控制；

（2）坦克都可以发射子弹，一旦击中对方游戏结束；

（3）如果坦克发生碰撞游戏结束。

和前面程序有所不同的是，这里我们将使用到两个子画面对象。当然，无论多少个都不是很要紧，因为我们使用的是便于扩充的面向对象思路来写子画面的。

我们已经多次提到过子画面程序的设计过程了，这里只将简单的介绍一下子画面游戏将分成的几大部分：

（1）构建子画面对象；

（2）第一次保存背景块；

（3）游戏主循环；

a. 检测子画面事件；

(a) 用户坦克事件(子弹的应用)；

(b) 计算机坦克随机事件；

b. 检测碰撞情况；

1. 构建子画面对象

这里想提一下从 pcx 内存向子画面内存读取所有子画面帧的问题。在子画面显示一节中我们已经介绍过如何读取一个子画面的所有帧了，那么要读取两个子画面应该怎么做呢？

我们看到第一个套画面坦克（8 个）被放在 pcx 图形的左上角(0,0)开始的第一行，而第二套子画面坦克则是在它的下面一行，也就是从图像内坐标(16,0)开始的 8 个边长为 16 的图块。在读取完第一个子画面之后只需要将读取起始位置调整到第二行开始就行了。具体的构建坦克两个子画面的函数如下：

```
void Sprite_Build(void) {
int index;
pcx_picture objects_pcx;
char key[2][7]={'8','2','4','6','5','q','0','w','x','a','d','s','q','z'};
PCX_Init((pcx_picture_ptr)&objects_pcx);//申请 pcx 图像内存
PCX_Load("attank2.pcx",(pcx_picture_ptr)&objects_pcx,1);//读取 pcx 图像到内存
//申请子画面对象空间
now=pre=(struct sprite *)malloc(sizeof(struct sprite_type));
head=now;//设定当前子画面对象为链表头
Sprite_Init(now,0,0,0,0,0,0,tank1,key[0]);//初始化子画面对象
//读取 pcx 图像内存中画面到子画面内存
for(index=0;index<MAX_SPRITE_FRAMES;index++) {
 PCX_Grab_Bitmap((pcx_picture_ptr)&objects_pcx,now,index,index,0);
}
//申请第二个子画面对象
now=(struct sprite *)malloc(sizeof(struct sprite_type));
pre->next=now;//第一个子画面对象的链表指针指向这个子画面对象
pre=now;
Sprite_Init(now,0,100,0,0,0,0,tank2,key[1]);
//读取第二套子画面
```

```
for(index=0;index<MAX_SPRITE_FRAMES;index++) {
 PCX_Grab Bitmap((pcx_picture_ptr)&objects_pcx,now,index,index,1); }
pre->next=NULL; //子画面链表指针尾指向空
PCX_Delete((pcx_picture_ptr)&objects_pcx); //删除 pcx 图像内存空间
}
```

这里还值得一提的是，我们在完成第一个坦克子画面对象的构建后，并将它设置成链表首，将它的下一个链表成员指向第二个坦克子画面结构体变量申请的空间。而在第二个坦克子画面构建完成后，我们就将它的下一个链表成员指向了空（NULL），这样就完成了最简单的单向链表的设置。

2. 第一次保存背景块

在完成构建子画面之后，需要在游戏主循环开始前完成所有子画面的第一次背景块的保存。对于一个子画面来说非常简单，我们只需要调用一个背景保存函数就可以了。而这里的子画面数量超过一个，我们如何做呢？

也许你会说我们写两个就可以了，这当然也可以。可是从面向对象的角度来说这样的方法将会对扩充带来极大的麻烦。倘若要扩充成红色警戒中的几十个坦克，就写几十个么？这似乎要犯“从三到万”的笑话了。

我们不是有链表么？让它发挥一下作用吧！

先前的程序中我们虽然构建和使用到了链表，但是还没有正式对它在游戏中的作用进行介绍。链表的最大优点就是链在其中的对象从头到尾一个都不会少。因为我们不需要知道到底有多少子画面对象，而只需要从链表头检验到链表尾就可以遍历所有的子画面对象。

基本的遍历方法如下：

（1）将指针指向链表头；

（2）对当前对象进行所需要的操作（比如保存当前对象背景块）；

（3）将指针指向下一个链表对象；

（4）判断当前对象是否为空（NULL），如果不是回到步骤 2，如果是则完成。

这种方法在子画面的背景块保存、事件检测和空间释放上都得到了应用。这里给出第一次保存所有子画面背景块的代码段：

```
now=head;
do{
    Behind_Sprite(now);//保存当前子画面对象背景
    now=now->next;//指向下一个字画面对象
}while(now!=NULL);
```

3. 游戏主循环

在先前所有的子画面程序主循环中只有检测子画面对象事件的任务，但在游戏中我们还需要检测碰撞情况。所以循环中包含了两个函数。以下给出游戏主循环代码段：

```
while(1){ Judge_Sprite();//检测子画面事件
    Judge_Died();//检测碰撞情况
}
```

4. 检测子画面事件

检测子画面事件的函数和前面所有程序中都是一样的。只是先前都只有一个子画面对象，而无法体现它的作用。这里我们便可以通过遍历链表对象的方法来检测调用所有子画面对象的事件。函数如下：

```
void Judge_Sprite(void) {
sprite_ptr sprite;
sprite=head;
do{     sprite->spr(sprite);
sprite=sprite->next;
} while(sprite!=NULL);
}
```

因为都使用遍历的方法，这段代码和第一次保存子画面背景块的代码非常相似。只是在定位当前对象后的具体对象操作上有所不同，前面是保存背景块函数而这里是调用当前对象事件函数。

由于有两个坦克子画面，当然也有两个事件函数：用户和计算机坦克事件函数。用户坦克事件函数只是在原来的子画面事件函数基础上增加了一个坦克射击键值的判断，而计算机坦克事件函数仅仅是修改了输入来源，用随机函数触发替代了用户输入。这两个函数将不再一一重复讲解了。

5. 子弹的应用

在这个游戏中子弹无疑是游戏的关键因素。那么如何将子弹加入程序中呢？

首先我们在子画面的结构中加入子弹结构体变量，使每个坦克子画面都拥有子弹，这样在两个坦克事件函数中就可以使用子弹了。以下给出子弹的使用规则：

（1）每个坦克在屏幕上只能出现一颗子弹；

（2）子弹的方向和坦克一致，子弹的起始位置处于坦克炮筒位置；

（3）当子弹越出屏幕（320*200）时，子弹作用消失；

（4）子弹击中对方坦克游戏结束；

（5）用户坦克子弹功能键为 0。

根据一上规则，我们在事件函数中增加了检测子弹功能键的代码和计算子弹数据的代码。这与子画面功能键检测和数据计算非常类似，在这里就不做详细代码分析了。

6. 检测碰撞情况

子画面碰撞情况的检测，其实就是对子画面之间的关系和相互影响进行判断。在这里碰撞情况有两种：

（1）坦克之间的碰撞；

（2）坦克和子弹的碰撞。

无论哪种导致的结果都是游戏结束。要检测这两种碰撞是否发生，首先需要获得两个子画面对象，然后对它们的相关值进行比较就行了。以下给出了检测碰撞的函数：

```
void Judge_Died(void) {
sprite_ptr tank1,tank2;
tank1=head;//取得第一个子画面对象
tank2=head->next;//取得第二个子画面对象
if(Sprite_Collide(tank1,tank2)||bullet_Collide(tank2,(bullet_ptr)&tank1->bullet)||
bullet_Collide(tank1,(bullet_ptr)&tank2->bullet)) { //检测碰撞
Sprite_Free();
Sheer();
Set_Video_Mode(TEXT_MODE);
exit(1);//发生碰撞了则游戏结束
} }
```

以下是判断子画面间碰撞的函数：

```
int Sprite_Collide(sprite_ptr sprite_1,sprite_ptr sprite_2) {
int dx,dy;
dx=abs(sprite_1->x-sprite_2->x);//两个子画面间横坐标距离
dy=abs(sprite_1->y-sprite_2->y);//两个子画面间纵坐标距离
//判断子画面间距离是否小于一个子画面身体
if(dx<(SPRITE_WIDTH-(SPRITE_WIDTH>>3))&&
dy<(SPRITE_HEIGHT-(SPRITE_HEIGHT>>3))) { return(1);//小于则返回碰到了消息
} else { return(0); } }
```

以下是判断子画面和子弹碰撞的函数：

```
int bullet_Collide(sprite_ptr sprite,bullet_ptr bullet) {
int dx,dy;
dx=abs(sprite->x+7-bullet->x);//子画面和子弹的横坐标距离
dy=abs(sprite->y+7-bullet->y);//子画面和子弹的纵坐标距离
//判断子画面和子弹距离是否小于一个子画面身体
if(dx*2<(SPRITE_WIDTH-(SPRITE_WIDTH>>3))&&
dy*2<(SPRITE_HEIGHT-(SPRITE_HEIGHT>>3))) {
return(1);//小于则返回碰到了消息
} else { return(0); } }
```

这里的子画面判断程序我们可以设计得更加好一些。因为当前的方法无法应付更多的子画面情况。倘若有 10 个子画面，我们就必须比较所有子画面之间的碰撞情况以及所有子

弹和所有子画面的碰撞情况。解决这个问题的办法并不能够简单依靠遍历的方法，而是使用类似于冒泡法的思路：

（1）使用遍历的方法，先取得第一个子画面对象；

（2）将它和所有其链表后的子画面和子弹（从下一个链表对象开始遍历）判别碰撞情况；

（3）循环 1、2 步骤，直到第一个遍历结束。

以下给出改进后的函数：

```
void Judge_Died(void) {
sprite_ptr tank1,tank2;
tank1=head;
do{ tank2=tank1->next;
   while(tank2!=NULL) { //将 tank1 和所有的其他子画面进行碰撞检测
      if(Sprite_Collide(tank1,tank2)||bullet_Collide(tank2,(bullet_ptr)&tank1->bullet)||
bullet_Collide(tank1,(bullet_ptr)&tank2->bullet)) {
            Sprite_Free();
            Sheer();
            Set_Video_Mode(TEXT_MODE);
            exit(1);
        } tank2=tank2->next;
   } tank1=tank1->next;//tank1 完成全部子画面碰撞检测后，被设置为下一个子画面
}while(tank1!=NULL);
}
```

事实上这有点类似于我们常常使用的两个嵌套的 for()语句，如同这里的冒泡算法使数组从大到小排列的代码段：

```
for(i=0;i<100;i++)
   for(j=i+1;j<100;j++)
       if(a[i]<a[j]) { temp=a[i];
           a[i]=a[j];
           a[j]=temp;
       }
```

以下给出一个简单坦克游戏例程。

程序功能：实现两架键盘控制坦克相互射击。

程序流程：

（1）初始化 320*40 的 pcx 图片内存空间；

（2）将 320*40 的 pcx 文件（8*2 个子画面都在这个文件中）调入该内存空间；

（3）初始化子画面空间（8*2 个子画面空间）；

（4）将内存中的图片按照 16*16 的大小对应复制给子画面空间；

（5）删除为 pcx 图片申请的内存空间；

（6）保存当前子画面位置和背景画面到内存；

（7）恢复刚才保存的子画面背景；

（8）如果有子弹在运动，则恢复刚才子弹位置及背景点颜色；

（9）判断是否有按键，如果没有则设置当前方向上的另一个子画面为当前子画面，并转到 11；

（10）如果有按键，则甲方：4 表示向左，6 表示向右，8 表示向上，2 表示向下，5 表示原地，1 表示射击，乙方：a 表示向左，d 表示向右，w 表示向上，x 示向下，s 表示原地，z 表示射击，q 表示退出程序；

（11）任意一个方向运动处理如下：

a. 标示当前子画面方向；

b. 计算相应的移动位置；

c. 计算当前应该调用的子画面图片序号（一个方向也有 2 个子画面）；

d. 如果有子弹射出，计算子弹初始位置及方向；

（12）如果有子弹在运动，计算其新的位置；

（13）保存新计算出的子画面位置及背景画面到内存；

（14）如果有子弹在运动，则保留新子弹位置和背景点颜色；

（15）如果有子弹在运动，则绘制新的子弹点；

（16）根据新的位置、当前子画面序号绘制子画面到屏幕对应位置；

（17）判断是否有子弹射击到子画面或者 2 个子画面是否相遇，如果相遇则退出程序；

（18）循环 7 到 17 步骤直到退出按键被触发；

（19）删除所有子画面空间。

主要函数：

```
int Get_Pixel(int x,int y); //取得屏幕点颜色
int bullet_Collide(sprite_ptr sprite,bullet_ptr bullet); //判断子弹是否击中对方
int Sprite_Collide(sprite_ptr sprite_1,sprite_ptr sprite_2); //判断子画面是否相遇
void tank1(sprite_ptr sprite); //用户坦克事件函数
void tank2(sprite_ptr sprite); //电脑坦克事件函数
```

程序要点：

（1）增加了子弹对象，虽然在屏幕上用一个像素表示子弹，而不是子画面。但是对它的动画效果的处理方法类似子画面处理：

a. 是否有子弹射出或者子弹在运动，无则继续判断；

b. 恢复上一个子弹点位置后面的点颜色；

c. 计算子弹当前的 x,y 位置；

d. 保存当前子弹点位置后面的点颜色；

e. 绘制当前子弹点；

f. 循环 a 到 e 直到子弹射出屏幕或者子弹射中目标。

（2）检测子弹是否射中坦克或者检测坦克是否相撞只需要将利用子弹和子画面结构中

的表明其当前位置的坐标变量之间的差值来判断，如果横坐标和纵坐标的差值都小于子画面的长、宽（16），那么说明射中或者相撞。

程序代码 tank.c 请查阅所附光盘的“source\10”目录。其效果如图 10-9 所示。

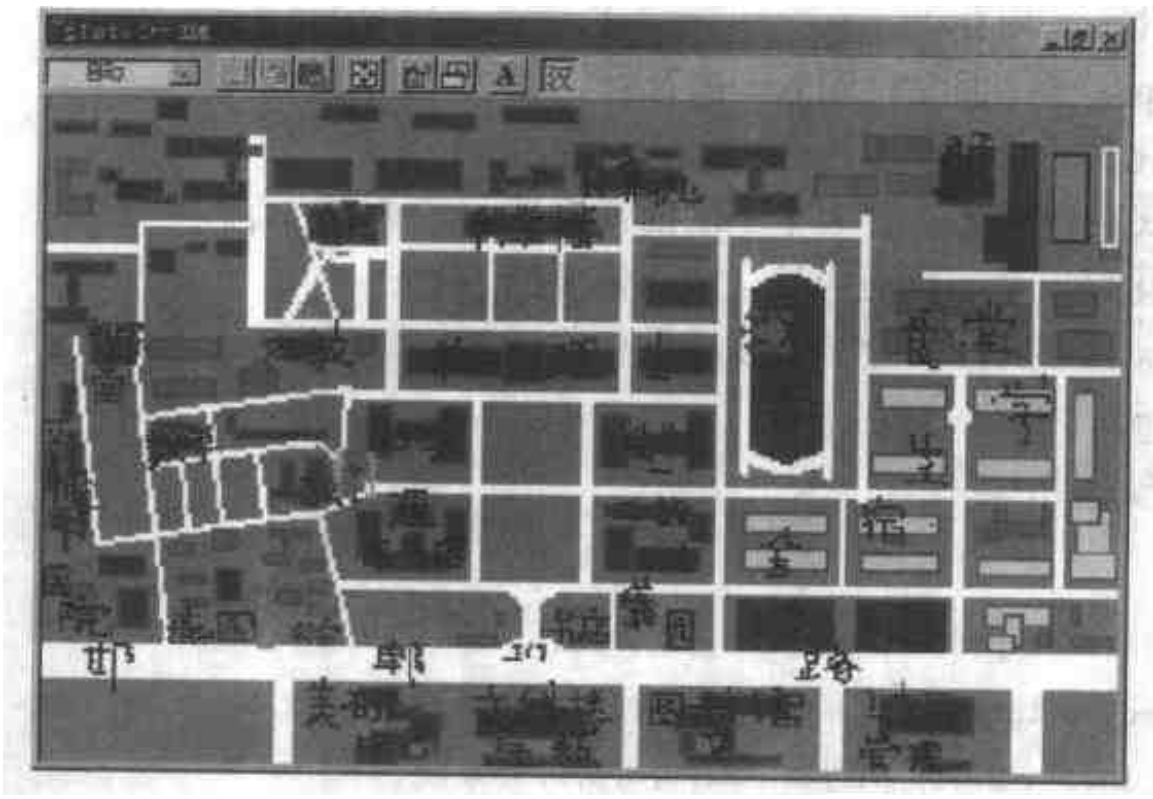


图 10-9　坦克游戏事件

10.6　子画面绘制

子画面的制作也是子画面游戏中非常重要的一部分。通常我使用在绘制游戏场景和子画面的时间和编程的时候差不多长，因为没有这些画面的绘制就好比“巧妇难为无米之炊”。更何况绘制子画面对于了解子画面技术中读取子画面到内存部分非常有帮助。

制作子画面图像可以利用以下软件：

（1）Windows 自带的画笔（当然你也可以使用 Photoshop 等图像制作软件）；

（2）ACDSee 看图软件，用于转换图形格式。

子画面绘制的过程如下：

（1）画笔→图像菜单→属性对话框。由于子画面通常是 16*16，这里将图像宽度设置为 16，高度也设置为 16，设置背景黑色；

（2）绘制出你要的第一个子画面；

（3）打开另一个画笔，设置宽度 320，高度 200，设置背景黑色；

（4）将子画面全选复制到 320*200 画面(0,0)开始坐标；

（5）将子画面先后翻转（图像菜单→翻转/旋转选项）出其他 3 个方向，并且都复制到 320*200 画面(16,0)、(32,0)、(48,0)开始的坐标；

（6）将子画面翻转回第一个方向，修改子画面使其和刚才子画面有相同方向上的不同和动画效果；

（7）按照前面的方法将它复制到 320*200 画面(64,0)开始坐标；

（8）之后和第 5 步骤相同，只是复制到的起始横坐标位置要从 64 依次增加 16；

（9）完成第一排 8 个子画面的绘制后，如果游戏需要，可以按照前面步骤从(0,16)坐标开始绘制第二排另一个对象的子画面；

（10）完成所有绘制后保存 320*200 的画面为 256 色的 bmp 文件；

（11）使用 ACDSee 将其转换为 pcx 文件。

有了 pcx 文件，程序将 pcx 文件读入子画面内存的过程如下：

（1）初始化 pcx 内存；

（2）读取 pcx 子画面文件到 pcx 内存；

（3）初始化第一个子画面内存；

（4）从 pcx 内存将第一行 8 个子画面依次读入子画面内存；

（5）初始化第二个子画面内存（如果需要的话）；

（6）从 pcx 内存将第二行 8 个子画面依次读入子画面内存；

（7）释放 pcx 内存。

以下是程序段调用过程：

```
...
PCX_Init((pcx_picture_ptr)&objects_pcx);
PCX_Load("attank2.pcx",(pcx_picture_ptr)&objects_pcx,1);
Sprite_Init((sprite_ptr)&tank1,0,0,0,0,0,0);
for(index=0;index<MAX_SPRITE_FRAMES;index++)
PCX_Grab_Bitmap((pcx_picture_ptr)&objects_pcx,(sprite_ptr)&tank1,index,index,0);
Sprite_Init((sprite_ptr)&tank2,0,100,0,0,0,0);
for(index=0;index<MAX_SPRITE_FRAMES;index++)
PCX_Grab_Bitmap((pcx_picture_ptr)&objects_pcx,(sprite_ptr)&tank2,index,index,1);
...
PCX_Delete((pcx_picture_ptr)&objects_pcx);
```

当然你对一个子画面对象的描述不一定只有 4 个方向 8 个子画面。我们经常看到一些游戏中子画面对象具有 8 个方向：上、下、左、右、左上、左下、右上、右下；而在一个方向上面通常坦克可以用 2 个画面实现动画，而类似于人物的画面可以使用 2 个也可以使用 3 个画面来实现更加逼真的动画效果。

320*200 的画面上，一行如果要画 12~16 个子画面是完全可以的，只是如果要画 8 个方向 3 个基本动作的 24 个子画面稍微需要想想其它办法。绘制画面其实并不困难，关键是你要清楚哪些子画面的方向，每对基本动作子画面序号是什么？这在以后程序的子画面调用中是非常重要的。

10.7　本 章 小 结

子画面技术是计算机游戏动画最实际的应用。子画面技术就是让屏幕中某些动画小对象调用图像文件，产生自身的动画和变换规则。具体地说就是将装有一系列子画面的图像文件调入内存数组，然后根据游戏当前进行的情况将其中的某个调入屏幕而产生动画。

子画面对象的动画过程：

（1）画整个屏幕画面；

（2）从文件保存若干子画面到内存；
（3）保存将要放入子画面位置的背景图像块到内存；
（4）将子画面放入屏幕画面中；
（5）接收游戏者信息或按照规则进行变量计算；
（6）将内存中保存的子画面背景图像块恢复到屏幕；
（7）处理游戏中的计算问题，并且根据游戏情况确定子画面放入的下一个新位置；
（8）循环 3~7 步骤，直到游戏结束。

我们首先必须定义子画面的结构，这里融入了面向对象的思路，其主要表现为：
（1）子画面的定义完善对象化；
（2）在子画面建立过程中强调面向对象概念；
（3）在游戏主程序中通过检测函数循环调用链表来检验每个子画面动画；
（4）为每个子画面都设计了自己的事件函数；
（5）在子画面对象死亡时，释放对象所有空间。

在子画面游戏制作之前我们必须首先绘制一系列个性化的子画面的图像，然后还要处理包括子画面保存与显示、背景保存和恢复、键盘控制和游戏状态检测、计算等问题。

学后建议

（1）尝试为子画面对象结构提供更多的属性和事件函数（包括增加鼠标控制子画面的功能）；

（2）考虑全部使用子画面技术来实现类似于 mario 的游戏，此游戏包括所有的场景都是由大小相同的子画面拼接而成的。

第11章　文件操作

本章导读

在游戏中文件的调用是非常普通和必要的。比如屏幕上的图形许多依靠调用图形文件来实现；而在游戏进入和退出的时候通常有调入和保存进度的选项，此时调用的是进度文件；在 RPG 等类型游戏中有大量的人物、地图、位置、对白、物品和各类属性，这些都是通过调用数据库类型的文件实现的。

对于游戏的进度和数据文件，我们最主要的操作就是保存和读取。而当数据内容非常多的时候建立一个有较好数据结构的数据文件也是非常重要的。因为它不仅可以方便我们对保存和读取的算法设计，同时也有利于进一步扩充和节约文件空间。

事实上，如果数据文件结构非常复杂、内容非常多的时候，建议使用数据库文件（例如 dBase 或者 FoxBase 文件）。因为这些专业的数据库软件不仅提供良好的数据批量输入途径，同时拥有经得起时间推敲的良好数据结构。我们只要学会如何在 C 语言游戏编程中读取数据库文件的内容就可以了。

对于文件操作其实我们在第 8 章已经介绍了图形文件的一些操作，本章将就普通文件的操作、游戏进度文件、简单数据文件和 dbf 数据库文件进行介绍。

本章重点

（1）游戏进度文件的保存和读取方法及其函数实现；
（2）游戏数据文件的结构设计和读取方法及其函数实现。

11.1　文件基本操作

文件的基本操作主要包括，文件的新建、打开、保存、关闭，以及文件指针的定位、读取和写入文件等操作。这些操作的相关函数在 C 语言标准库函数头文件 stdio.h 中都有定义。

11.1.1　建立、打开和关闭

C 语言提供了文件打开和关闭的函数 fopen()与 fclose()(见所附光盘的“book\附录 D”)，以下是一个打开和关闭文件的例程 file.c：

```
#include <stdio.h>
int main(void) {
   FILE *in, *out;
   if ((in = fopen("test1.txt", "rt"))//以读的方式打开一个文件
   == NULL) {
  fprintf(stderr, "Cannot open input file.\n");
  return 1;
```

```
 } fclose(in);//关闭文件
 if ((out = fopen("test2.dat", "wt"))//以写的方式打开一个文件
 == NULL) {
fprintf(stderr, "Cannot open output file.\n");
return 1;
 } fclose(out);//关闭文件
 return 0; }
```

通常新建一个文件实际上就是使用 fopen()函数的"w"方式或者"w+"方式。此外在文件调用结束后，建议尽快使用 fclose()函数关闭文件，以便下一次调用。因为如果在没有关闭文件的情况下第二次使用 fopen()函数打开文件会造成失败的。

11.1.2　读取和写入

在对文件内容进行操作的过程中，我们将涉及到 3 个动作：指针定位、读取和写入文件。它们分别对应 3 个 C 语言标准库函数 fseek()、fread()和 fwrite()（见所附光盘的“book\附录 F”）。以下给出读取和写入文件的函数例程 rw.c：

```
#include <string.h>
#include <stdio.h>
int main(void) {
   FILE *stream;
   char msg[] = "this is a test";
   char buf[20];
   if ((stream = fopen("test3.txt", "w+"))
   == NULL) {
  fprintf(stderr, "Cannot open output file.\n");
  return 1; }
   fwrite(msg, strlen(msg)+1, 1, stream); //写一些数据到文件
   fseek(stream, SEEK_SET, SEEK_SET); //从文件头开始计算偏移
   fread(buf, strlen(msg)+1, 1, stream); //读取文件中的数据并且显示
   printf("%s\n", buf);
   fclose(stream);
   return 0; }
```

11.2　游戏进度文件

在玩大点的游戏的时候经常会碰到一次无法将游戏进行到底的情况，遇到这种问题我们就需要将目前游戏进行的状态保存，然后等下一次玩游戏的时候再调出来。所以许多游戏都有“save”和“load”两个菜单功能供我们选择。

事实上，我们保存的内容就是游戏的进度，而保存和读取的文件被称为进度文件。在

进度文件中通常保存的是游戏进程中的一些游戏进度和状态变量，例如，游戏进行到的位置、情节发展到的等级、游戏主人公拥有的物品和功力等。

这些进度数据有一个特点，通常其内容都很少，可能就是几个数字和几个名词。所以对它们的处理与对数据库数据的处理是不同的。应该说进度文件的处理比数据库文件的处理简单了很多。下一节我们将提出两种进度文件读取和保存的办法。

11.2.1　两种方法

对进度文件的两种处理方法分别被称为定位处理法和分割符法。要了解这两种方法我们可以先来看一下两个进度文件的数据：

第一个文件（定位处理法的进度文件）：

```
2  11
```

第二个文件（分割符法的进度文件）：

```
2;11;
```

从两个进度文件可以看出，它们都是保存了两个数据。为了明确是两个数据，前者给每个变量设定了 4 位的空间；后者使用了分号作为分隔符号。

要实现前者的读取和保存非常简单，我们需要使用 C 语言标准函数库的文件读取和写入的函数 fprintf()与 fscanf()（其定义见所附光盘的“book\附录 F”）。以下给出其实现定位处理法对进度文件保存和读取的例程 read.c：

```
#include<stdio.h>
void main(void) {
   FILE *game_file;
   int xa=2,ya=11;
   game_file=fopen("game.dat","wt");//打开第一个文件 game.dat
   fprintf(game_file,"%4d",xa);//向文件写入数字 xa，占用 4 个位置
   fprintf(game_file,"%4d",ya); //向文件写入数字 ya，占用 4 个位置
   fclose(game_file);//关闭文件
   xa=0;
   ya=0;
   game_file=fopen("game.dat","rt");
   fscanf(game_file,"%4d",&xa);//从文件读取数字 xa，读取 4 个位置
   fscanf(game_file,"%4d",&ya); //从文件读取数字 ya，读取 4 个位置
   fclose(game_file);//关闭文件
   printf("xa=%d,ya=%d",xa,ya); }
```

用定位处理法实现一个进度游戏非常方便。一个简单的游戏例程 game1.c 请查阅所附光盘的“source\11”目录。

定位处理法对于所有进度数据我们只需要在写入进度文件时给每个数据留出足够的固定空间就可以了，在读取的时候按照写入时所分配空间大小依次读取。这种方法的优点主要在于：

（1）使用简便：fscanf()和 fprintf()函数的使用很简单；

（2）定位方便：读取时只需要按照写入时的定位就可以了；

（3）类型对应：在写入和读取文件时可以根据对应的类型直接在文件和变量中传递。

缺点：

（1）对文件中字符和字符串读取存在一定问题；

（2）对不确定长度变量空间分配有一定难度。

针对定位处理法通用性和灵活性不够的问题我们提出第二种进度文件实现方法，也就是前面说到的分隔符法。分隔符法只关心分隔符而不关心变量占用空间长度，碰到一个分隔符就是一个变量，但是这种方法也有缺点，在变量类型方面需要我们自己做转换。从下一节开始我们将详细介绍这种方法。

11.2.2　保存进度文件

有些游戏由于进度相对较慢或者过程比较长，玩家通常无法一次性玩到结束。于是，游戏会提供保存和读取上次保存内容的功能。这里就会涉及到文件的读写功能。

通常需要保存的会是屏幕上各类对象的位置、当前图像和各类游戏当前数据。这个保存过程通常可以理解为将程序中的部分变量保存到进度文件，当我们重新开始游戏要调用这些数据的时候则又可以理解为将文件读取到变量中。

对于进度文件的保存结构，通常建议越简单、越直观越好。如果只有单个数据变量，写入文件就完事了；如果是多个数据变量，我们只需要在它们之间加一个分割符号就可以了，例如：“32,00,32,09,”；进度文件中很少碰到大量数据的输入，即便如此通常也可以使用分割符号的方法。

使用分割符号的方法有许多好处：

（1）便于保存和读取；

（2）使数据文件更加紧凑、直观；

（3）自选符号可以灵活调整，可以避免和数据重复，比如，原本准备用“空格”作为分割符号，但是进度变量中有“空格”作为数据，那么就选中“分号”、“逗号”或者其它未被用到的符号。

常用分割符号有“逗号”、“空格”、“分号”、“回车”，但通常不建议使用“空格”和“回车”。

这里以一个最简单的进度游戏作为例子：屏幕上有一个小的长方快，可以使用上下左右键对其进行移动，按“q”键之后可以保存当前位置以备下次游戏调用。

在这个游戏中，所谓的进度实际上就是长方块在屏幕中的对应于 x,y 坐标轴的索引值 X,Y。假设长方快大小为 16*10（屏幕为 320*200），长方块的初始位置为屏幕的作上角(0,0)，用索引值变量 X=0、Y=0 表示为(X*16,Y*16)。当我们按下右键的时候长方快的 X 索引值随即加 1，其对应坐标变为(X*16,Y*16)->(16,0)。此时如果保存游戏，我们只需要将 X,Y 两个变量对应值 1，0 保存到文件就可以了。

在保存的过程中，我们建立一个文件叫 load.dat。将 X 变量写入文件第一个位置，然后写入一个“逗号”来分割，之后写入 Y 变量，关闭文件就实现了进度文件的保存。

这里我们发现，在保存进度变量的时候，必须采用一个分割标志才能判别多个变量。

当然我们还有一个方法，就是稍后要说的可以通过确定每个变量在文件内的固定起始位置来区别它们。

保存进度变量的过程可以分为以下步骤：

（1）新建或者打开进度文件；

（2）定位到文件头（通常打开文件的时候已经在文件头了）；

（3）将当前变量写入文件；

（4）写入逗号；

（5）找到下一个要保存的进度变量；

（6）重复第 3、4、5 步骤，直到所有进度变量都保存到文件；

（7）关闭进度文件。

以下的程序 save.c 模仿该游戏保存 X,Y 变量到文件的过程：

```
#include<io.h>
#include<stdio.h>
#include<dos.h>
#include<string.h>
#include<math.h>
#include<stdio.h>
#include<bios.h>
#include<mem.h>
#include<fcntl.h>
#include<stdlib.h>
#include<conio.h>
void main(void) {
    char flag;
    char *flags;
    char *file
    FILE *stream;
char *x="2",*y="13";//要保存的进度变量
flag=',';//分割符，这次是逗号
flags[0]=flag;
file="load.dat";//进度文件名称
if ((stream = fopen(file, "w+b")) == NULL) { //新建或者打开进度文件，写方式
     fprintf(stderr, "Cannot open output file.\n");
     return 1;      }
fwrite(x,strlen(x),1,stream);//写入第一个进度变量
fwrite(flags,1,1,stream);//写入分割符号
fwrite(y,strlen(y),1,stream);//写入第二个进度变量
fwrite(flags,1,1,stream);//写入分割符号
```

```
fclose(stream); } //关闭文件
```

此时，打开 load.dat，你可以看到里面的内容是“2,13,”。对于前面那个进度游戏，2 表示长方块在横坐标位置上的第 3 格，即横坐标 32；13 表示长方块在纵坐标位置上的第 14 格，即纵坐标 172。于是我们就可以把点(32,172)作为长方块左上角在屏幕上重新画出长方块刚才保存时候的位置。这里值得一提的是如果没有“逗号”的分割，我们将无法理解这里到底是 2,13 还是 21,3...

在此基础上，我们考虑建立一个保存进度文件的函数以便实现一次性将对游戏所有进度变量进行保存。为此，我们考虑将所有进度变量串连起来组成一个数字型的数组，并且以“-1”作为结束符号。不过这个函数只能保存不确定长度的数字型进度变量。以下给出一个具体的保存进度数据到文件的通用函数（这里的进度文件都是数字型变量）：

```
void Write_To_File(char flag,char *file,int *word) {
    const char *xy;
    int i=0;
    FILE *stream;
    if ((stream = fopen(file, "w+b")) == NULL) { //新建或者打开一个进度文件，写方式
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    } while(word[i]!=-1) { //直到字符串结束符"-1"
    itoa(word[i],xy,10);//将字符串型变量转化成数字形
    fwrite(xy,strlen(xy), 1, stream);//写入这个进度变量到文件
    fputc(flag,stream);//写入分割符号
    i++;
    } fclose(stream); }
```

当主程序要保存进度数据到文件时，只需要以如下方法一次调用函数即可保存进度文件：

```
int *word;
char flag=',';
...
word[0]=a;      //a 为进度变量 1
word[1]=b;      //b 进度变量 2
word[2]=c;      //c 进度变量 2
word[3]=-1;     //设定结束标志
Write_To_File(flag,file,word);//调用进度保存函数
```

11.2.3　读取进度文件

如何将文件内的“2,13,”读回 X,Y 变量呢？

读取进度文件的过程如下：

（1）打开进度文件；
（2）定位到文件头（通常打开文件的时候已经在文件头了）；
（3）读取当前开始的每个字符，直到出现“逗号”；
（4）从本次读取开始的位置到逗号前取出变量；
（5）定位文件指针到“逗号”后一个位置；
（6）重复第 3、4、5 步骤，直到所有逗号读完；
（7）关闭进度文件。

以下给出一个一次性读取进度文件所有进度变量的通用函数：

```
void Read_From_File(char flag,char *file,int *word) {
    int i,j=0,k=0;
    char *buf;
    FILE *stream;
    if ((stream = fopen(file, "r+b")) == NULL) { //以读的方式打开进度文件
        fprintf(stderr, "Cannot open output file.\n");
        return 1;      }
    for(i=0;!feof(stream);i++) {
        fread(buf,1, 1, stream);//读取第一个字符
        if(buf[0]==flag)//判断是否是分割符
        {//如果是则读取当前分割符号前面的数据（到前一个分割符号之间的）
            fseek(stream,k, SEEK_SET);
            fread(buf,i-k+1,1,stream);
            buf[i-k]='\x0';
            word[j]=atoi(buf);
            j++;
            k=i+1; } //当前分割符数字加 1
    } fclose(stream); }
```

此函数每次调用可以取得任意一个进度变量，主程序调用过程如下：

```
    Read_From_File(flag,file,word);//读取进度文件
    a=word[0];//将第一个进度变量从 word 中取出
    b=word[1];
    c=word[2];
```

这里还提供了一个可以从文件读取第 n 个进度变量的通用函数：

```
char *fread_char(char flag,char *file,int num) {
    int i,j=0,k=0;
    char *buf;
    FILE *file;
    if ((stream = fopen(file, "r+b")) == NULL) {
```

```
        fprintf(stderr, "Cannot open output file.\n");
        return 1; }
    for(i=0;!feof(stream);i++) {
        fread(buf,1, 1, stream);
        if(buf[0]==flag) {
            j++;
            if(j==num) { //直到第 num 个进度变量才读取
                fseek(stream, k, SEEK_SET);
                fread(buf,i-k,1,stream);
                buf[i-k]='\x0';
                fclose(stream);
                return(buf);
            } k=i+1;
    } } fclose(stream); }
```

用第二种方法读写进度文件的游戏 game2.c 请查阅所附光盘的“source\11”目录。

11.3　游戏数据文件

相对于进度文件来说，游戏数据文件更加复杂一些。这类文件通常被用于 RPG 文件中。比如有大量的人物对话、大量的地图调用和地图位置数据以及物品和属性。这些数据不同于进度文件中的数据，进度文件中的数据只是用于保存记录当前游戏进度状态数据，而数据文件内的数据则是贯穿游戏整个过程的，而且通常都不进行改变（只读）。

从另一个角度来说，数据文件的数据内容也远远大于进度文件中的数据。所以，对于数据文件的写入方法通常会使用其它专门的文本软件或者数据库软件。而在读取的时候也采用不同于读取进度文件的读取方法。

首先有必要谈一下简单数据文件的保存结构。上一节提到使用分割符号将数据分割开来的方法，这种方法主要是针对少量的非重复类型的数据。而简单数据文件需要放置大量的类型重复的数据，其不仅需要分割单个数据还要分割类型重复的数据群。所以，单单使用分割符号的方法将会给读取带来非常大的麻烦，同时也是非常低效、慢速的。

在数据文件中通常都固定每个单元格的空间大小，而每行记录单也都确定了空间大小。数据文件可以将分割符号和固定记录单起始位置的方法来解决数据存放问题。比如，游戏中的对话数据包括对话人、对话内容，其保存结构如下：

(文件头)至尊宝;师傅;唐僧;徒弟;至尊宝;去哪里呀师傅？;唐僧;走;唐僧;天竺;=

(文件第 112 个字节开始)至尊宝;仙子;紫霞仙子;猴头;至尊宝;我会用七彩云来接你;紫霞仙子;不许骗人;至尊宝;sure;=

(文件第 224 个字节开始)猪八戒;猴哥;至尊宝;猪头;猪八戒;月光宝盒在哪里;至尊宝;不知道;=

…

可以看出在数据结构中，类似于记录单的数据段都有固定的文件内起始位置，同时它们也都有一个结束符号，在这里是“=”；而数据群内相当于单元格的数据还是使用分割符号来区别。两种方法的结合大大节约了文件的开销和降低了读取函数的工作强度。比如，要第二个“记录单”开始的数据群，只要将文件指针定位到 112 个字节开始就可以了；然后读到一个逗号处理一个“单元格”；一直读到“等号”的时候“记录单”结束。

有一点需要注意的是，数据群之间通常是固定字节数的，比如这里就是 112 个字节 1 个数据群。那么，如何确定数据群之间的单位字节数呢？可以通过计算最长的数据群长度再预留一些空间来实现。这样将比较节约数据文件开销。

读取此结构的函数可以通过修改进度文件任意进度变量读取函数 fread_char()来实现，以下给出读取某个具体数据单元的函数：

```
char *fread_char_seek(char flag,char *file,int num,int seek,char end) {
    int i,j=0,k=0;
    char *buf;
    FILE *stream;
    if ((stream = fopen(file, "r+b")) == NULL) { //以读方式打开进度文件
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    } fseek(stream,seek,SEEK_SET);//到当前数据段
    for(i=0;!feof(stream);i++) {
        fread(buf,1, 1, stream);
        if(buf[0]==end) { //判断是否到数据段结束
            buf[1]='\x0';
            fclose(stream);
            return(buf); }
        if(buf[0]==flag) { //判断是否到数据项结束
            j++;
            if(j==num) { //读取第 num 个数据项
                fseek(stream, seek+k, SEEK_SET);
                fread(buf,i-k,1,stream);
                buf[i-k]='\x0';
                fclose(stream);
                return(buf);
            } k=i+1;
    } } }
```

如果每次要读取一段对话并且显示，则可以使用函数：

```
void Read_Words(int x,int y,char flag,FILE *stream,int num,int seek,char end) {
    char *answer;
```

```
for(num=1;answer[0]!=end;num++) { //从当前数据段的第一个数据项到最后一个数据项
    answer=Read_Char_Seek(flag,stream,num,seek,end);//依次读取当前数据项
    if(answer[0]!=end) {
        hz24_k(x,y,answer,color);//显示读取数据项
        getch();
        Fill_Screen_Size(0,x,y,320,y+25);//清屏
    } };
```

一个读取数据文件的函数和程序 xiyouji.c 请查阅所附光盘的“source\11”目录。

这里还值得一提的是，如此大量的数据采用什么方法输入数据文件内呢？建议不要使用程序写入的方法，虽然上一节进度文件的写入是用编程的方法实现的。但对大量的不同类型数据的写入事实上非常类似将数据加入数据库。如果要用程序完成它，必须建立一个相当完善的数据库各类操作函数(这在稍后会讲到)。这里我们建议使用文本编辑器来写入数据。

UltraEdit-32 是一个非常好的文本编辑器，在写入数据的时候可以采用“十六进制”观看模式，从而直观的了解当前在文件内的字节位置，将非常简单、明了地实现对于固定起始字节的数据群写入操作。这里的 xiyouji.dat 数据就是利用 UltraEdit-32 写入的。

11.4　dbf 文件

DOS 下最常用的关系数据库是 dBase 和 FoxBase。如果能够使用 C 语言程序将其.dbf 文件数据直接读出，对于我们设计需要数据文件的游戏将带来很大的方便。

Dbf 数据库文件比我们自己的数据文件有两个优点：

（1）结构固定，无需设计；

（2）存储方便，无需文件内地址搜寻再写入；

用 dbf 来保存游戏数据文件绝对比我们自己干省力很多。只是目前 dbf 数据库文件可以说也已经走向末路，估计很少有人还在用它。所以这里只是做一个简单的介绍。

11.4.1　dbf 文件结构

dbf 文件的结构主要分为头文件区和数据区。头文件区包括数据库参数部分和字段说明部分。数据库参数部分固定长度为 32 个字节。其主要内容如表 11-1 所示。

表 11-1　dbf 头文件区

偏移（字节）	保存内容
0H	03H，暨数据库文件标识
1-3H	文件最后改变时间（年月日）
4-7H	数据库记录数
8-9H	头文件区长度
A-BH	记录长度

字段说明部分占用字段数乘以 32 个字节的空间，每个字段 32 个字节内容如表 11-2 所示。

表 11-2　dbf 文件字段说明

偏移（字节）	保存内容
0-AH	字段名称
BH	字段类型
10H	字段长度
11H	小数位数

文件头以“0DH”（FoxBase）或者“0DH,00H”作为头文件结束标志。

数据区按照所有字段给定长度以记录顺序存放。每条记录以 20H（未删除标识）或者 2AH（删除标识）开始。数据区结束位置（也就是 dbf 文件结束位置）有 1AH 作为标志。

dbf 数据库文件和上节的数据文件的主要区别表现在：

（1）dbf 文件有头文件，数据文件没有；

（2）dbf 文件中所有字段（对应于：单元格或者每条数据）都有固定的字节大小，数据文件不确定大小只是以“逗号”分割；

（3）dbf 文件通常文件大小远远超过数据文件；

（4）dbf 文件的维护比数据文件容易很多；

可见两者各有利弊，dbf 文件在输入数据和查看数据的时候有相当的优势；而数据文件结构非常简单且文件大小远远小于 dbf 文件。

11.4.2　dbf 文件读取

在了解 dbf 文件结构后，我们发现 dbf 文件数据的读取其实是非常简单。其主要读取方式如下：

（1）读取数据库参数部分（最精简）：

a. 读取日期；（可省略）

b. 读取记录数；（可省略，游戏使用中通常不省略）

c. 读取头文件区长度；（可省略，用头文件区结束标识 0DH 或者 0DH，00H 也可以定位）

d. 读取每条记录长度；（可省略，用所有字段长度加上删除标识可以计算出）

（2）读取字段说明部分：（游戏中最重要的是读取内容）

a. 读取每个字段的名称；

b. 读取字段类型；

c. 读取字段长度；

d. 读取小数位数；（可省略）

（3）读取数据区：

a. 读取每个记录的删除标志；（可省略）

b. 读取每个字段单元；（游戏中最重要是读取内容）

一个简单的 DBF 文件读取例程 dbf.c 请查阅所附光盘的“source\11”目录。

11.5　本 章 小 结

在周期较长的游戏中提供保存功能是非常必须的，因为你不可能要求游戏者通宵达旦地将游戏玩到底。此时我们必须将游戏中与进度相关的数据保存起来，比如游戏对象的位置与目前的状态、等级和拥有的物品等。这些数据都比较简单可以线性地保存到文件中，只要固定每个数据长度或者使用间隔符号来将它们分割开来就可以了。我们可以使用两种方法来保存这些数据。第一种方法使用 fscanf()和 fprintf()函数固定长度来实现写入和读取；第二种方法通过使用分割符号的函数来完成。前者实现函数非常简单，但是数据必须固定长度，后者就没有长度限制了。这里提供的第二种分割符函数保存和读取的都是数字型变量，如果要支持更多类型请读者自己修改。

RPG 游戏需要使用大量的对话、实物、人物状态和地点数据，这些数据都需要通过调用数据文件来实现。我们可以针对不同的游戏数据情况设计对应的数据保存结构，通过简单的文本处理软件将这些数据写入文件。本章提供了读取数据文件的函数，读者可以在它的基础上建立适合的游戏函数。数据文件由于非常大，通常除了数据间的分割符以外还需要对整段的数据固定长度，从而可以高效地读取。

我们还可以使用一些简单的数据库软件（比如 FoxBase、dBase）来将制作数据文件，dbf 文件无疑是常用的数据文件格式。由于 dbf 文件的结构非常简单，我们在掌握其结构后便可以通过 C 语言程序调用它了。

学后建议

（1）制作改写游戏数据文件的通用函数，制作改写 dbf 类型文件中单元数据的通用函数；

（2）看一下附录 A 中简单数据库软件的设计，考虑设计一个较为通用的数据软件；

（3）研究一下目前较为流行的数据库软件（比如 VF、PowerBuilder）文件的结构，考虑一下是否可以使用 C 语言来调用。

第12章　声音技术

本章导读

真正好的游戏不仅需要在游戏设计思路上花功夫，事实上更多时候要看游戏的多媒体效果。多媒体效果主要包括了美工、音效等，前面我们着重介绍了如何使游戏屏幕更加生动和丰富多彩，本章我们将对于游戏中声音的实现进行一些探讨。

DOS 游戏中往往对声音不太重视，以至于许多游戏根本没有声音，当然这也是受到当时计算机发展限制的。在 C 语言环境下人们比较熟悉的是调用 PC 喇叭发声，然而 PC 喇叭的表现能力实在太差了，并且会影响游戏的运行，此时我们碰到了两个难题：

（1）如何在 DOS 下实现背景音乐，使游戏和音乐能够同时互不干扰对方的顺利运行？

（2）是否可以让 C 语言来驱动音箱发声，来播放 Windows 下流行的 wav 音效文件？

在我看来这两个问题是 C 语言游戏编程中和大内存调用问题一样可并列为三个难点，本章将尝试解决声音方面的问题。

本章重点

（1）直接调用硬件端口实现扬声器发声，及通过改写时间中断实现扬声器背景音乐播放；

（2）声卡的内部结构、编程接口、wav 文件结构，C 语言实现 wav 声音文件声卡播放的思路和程序实现；

（3）通过改写时间中断和 wav 文件播放程序，实现声卡背景音乐播放的办法；

12.1　PC 喇叭发声

PC 喇叭的发声函数 sound(frequency)，其中 frequency 是发声的频率；另外一个 nosound()函数用来消除声音。而声音持续的长度由延迟函数 delay(milliseconds)来实现；相关函数请查阅附录 D。

一个最简单的发出声音的方法是：

```
sound(frequency);//发声
delay(milliseconds);//延迟一定时间
nosound();//关闭扬声器
```

12.1.1　发声系统

我们可以尝试对硬件端口进行操作来实现 sound()和 nosound()函数的功能。扬声器实际上是通过对其输入寄存器分配的 I/O 端口 60h 和 62h，输出寄存器控制端口 61h 进行操作实现发声的。其中对于控制端口 61h 当其低 2 位如果同时是 1 表示允许发声（位 0 控制定时器驱动扬声器，位 1 控制扬声器电路开关），当位 1 为 0 则关闭扬声器电路而无法发声。

在允许发声之前我们必须首先设定好发声的频率，这是针对 2 号定时器端口 43h 和 42h

进行的。首先我们要向 43h 端口写入 0b6h，初始化定时器的方式寄存器，此时定时器 2 等待接收计时常数。之后在 42h 端口写入一个常数（2 个字节），此常数实际就是 123280H（533H*896Hz）除以我们要产生的声音频率。

以下是发声函数：

```
void Sound(int freq) {
asm{
mov    al,    0B6h
out    43h,   al//初始化定时器
mov    dx,    12h//除数高位
mov    ax,    533h*896//除数低位
div    freq//除数频率
out    42h,   al//常数低位写入端口 42h
mov    al,    ah
out    42h,   al//常数高位写入端口 42h
inal,    61h//从端口 61h 读入控制数据
oral,    3//低 2 位置 1
out    61h,   al//输出到端口 61h，发声
} }
```

以下是关闭扬声器函数：

```
void Nosound(void) {
   asm{
   inal,    61h//从端口 61h 读入控制数据
   xor    al,    3//低 2 位置 0
   out    61h,   al//不允许发声
   } }
```

我们发现发声函数、延迟函数和关闭扬声器函数常常是被一起使用的。那么我们完全可以写一个函数将它们都放进来，这样在调用的时候可以方便很多。以下是发声延迟函数：

```
void Sound_All(int freq,int clicks) {
Sound(freq);//发声
Delay(clicks);//精确延时
Nosound(); }//关闭扬声器
```

12.1.2　PC 喇叭播放歌曲

如果可以在游戏的片头、片尾或者过渡中加入一些音乐歌曲可以让游戏大大的增色。我们可以通过将一些声音的频率和音乐中的音符对应起来，有关音阶和频率的对应关系对照如表 12-1 所示。

表 12-1　音阶与频率对照表

	C	C+	D	D	E	F	F+	G	G+	A	A+	B
LOW	131	139	147	156	165	175	185	196	208	220	233	247
MIDDLE	262	277	294	311	329	349	370	392	415	440	466	494
HIGH	523	544	587	622	659	698	740	784	831	880	932	988

一个音阶、音符循环播放程序 cdefab.c 请查阅所附光盘的“source\12”目录。

CDEFGAB 音阶频率实际就是 C 音阶下 7 个音符的频率，所以前面的音阶和频率对照表也可以写成低、中、高 C 音阶每个音符和频率的对应关系表 12-2。

表 12-2　C 音阶频率对照表

	1	2	3	4	5	6	7
LowC	131	147	165	175	196	220	247
NormalC	262	296	330	349	392	440	494
HighC	523	587	659	698	784	880	988

程序演奏《渴望》这首歌曲的具体代码 song.c 请查阅所附光盘的“source\12”目录。

12.1.3　扬声器背景音乐

在游戏中我们使用的声音必须是不影响游戏本身运行的，也就是通常所说的背景音乐。如果我们将 Sound(freg)和 Delay(time)函数连用必定是占用前台而使游戏停止下来，这是谁都无法接受的。于是在 Windows 下的解决方法非常简单，因为 Windows 本来就是多任务环境；可是 DOS 不是多任务环境，我们该怎么做呢？

我们需要考虑改变原先的一个声音播放过程：

```
Sound(freq);//发声
Delay(clicks);//精确延时
Nosound();//关闭扬声器
```

要实现背景播放音乐关键是将延迟函数 Delay()完全占用的系统时间还给计算机，同时还要保证声音的延迟。我们可以这样做：

```
Sound(freq);//发声
…//开始计算时间中断次数
…//这里做其它任何你想做的事情
…//根据延迟时间换算时间中断次数足够了
Nosound();//关闭扬声器
```

发现 DOS 下的这个时间中断是可以为我们所用的，它每秒钟产生 18.2 次，于是我们可以通过改写它来实现背景音乐。具体就是在这个中断函数中要做以下几件事情：

（1）设置累加变量；

（2）调用 Sound(HZ[])函数,具体调用 HZ[]中哪个数组要根据累加数来决定；

（3）当累加数和每个音符要求延迟时间相同的时候调用 nosound()函数（此时累加数清零），实际上就是完成 delay()的工作。

比如要对一个音符发声延迟 500 毫秒，原本我们是这样做的：

```
Sound(freq);
Delay(500);
Nosound();
```

现在我们则省略了 Delay()函数，在中断函数调用 Sound()函数开始进行变量累加，一直加到 500/1000*18.2=9 的时候（也就是之后第九次调用中断函数），调用下一个 Sound()函数。这样在期间 9 次中断函数时扬声器时钟打开并发声，但这一切完全不影响 9 次中断期间前台的游戏操作，因为我们只需要做第一次前打开扬声器，第九次后重新改变扬声器发声内容和每次中断累加的操作。

以 PC 扬声器实现背景音乐的例程 soundgd.c 请查阅所附光盘的“source\12”目录。

12.2　声 卡 技 术

估计很少人会在 C 语言中使用声卡，可是所有人都不能接受在游戏中仅仅只能听到扬声器那乏味的声音。事实上 C 语言中调用声卡要远远比 Windows 编程要难，这其实也给了我们一个了解声卡的机会。

声卡其实是将自然界的声音进行数字化，并且将它再用数字化的方法通过音像展现在我们的耳边；这正如同显卡处理数字化的画面并且通过屏幕让我们的眼睛看到一样。声卡最重要的特点也就是将模拟信号转化成数字信号（A/D）和将数字信号转化成模拟信号（D/A）。前者是在进行录音采样的时候，后者是在进行播放的时候。

DOS 下最流行的声卡是创新公司（Creative Lab）的 Sound Blaster 系列。大量 DOS 下游戏软件都是使用创新公司提供的 CT-VOICE 驱动程序来播放 VOC 格式的声音文件。但是由于 VOC 格式在现在电脑中不再流行了，稍后我们将不介绍这种格式声音文件的播放技术。

声卡主要包括 DSP 芯片、CODEC 芯片、控制芯片、FM 合成芯片和混音器芯片。这里将只介绍和我们编程相关的 DSP 芯片。

12.2.1　DSP 简介

DSP（DIGITAL SOUND PROCESSOR）芯片是用来对声音进行输入和输出的，所有的数模相互转换（A/D 或者 D/A）都在这个芯片中进行。稍候介绍的 wav 格式的声音文件就是通过 DSP 芯片机进行录制和播放的。表 12-3 是 DSP 芯片的端口地址。

表 12-3　DSP 芯片端口属性

端口地址	端口功能	读写属性
2X6H	复位重置 DSP 端口	写
2XAH	数据输入端口	读
2XCH	命令、数据输出端口和数据接收缓冲状态端口	写和读
2XEH	DSP 数据是否可读	读

12.2.2　DSP 端口寻找

我们必须首先找到声卡 DSP 芯片的端口基地址才能对它进行具体的操作。找端口实际和复位 DSP 端口很有关系，以下来介绍一下入复位 DSP 端口：

（1）向复位端口写入 1；

（2）向复位端口写入 0；

（3）等待数据可读端口（2XEH）第 7 位变成 1；

（4）延迟 100 毫秒；

（5）询问数据输入端口是否得到 AAH，如果是就是复位成功（找到端口的标志也是这个）。

对于寻找端口我们只需要从声卡端口最小可能基地址 210H 开始，循环进行以上的复位端口工作，每次循环后将基地址增加 10H，直到得到 AAH 就是 210H+X*10H 端口了，如果一直到 260H 都无法找到则说明没有安装声卡。

以下是寻找 DSP 端口的函数：

```
void Testsb(){
    Cnt1=NumberOfTimes1;
    while((Port<=0x260)&&!Found) {//查找声卡端口
        outportb(Port+0x6,1);//向复位端口写入 1
        outportb(Port+0x6,0);//向复位端口写入 0
        Cnt2=NumberOfTimes2;//#define NumberOfTimes2 100
        while((Cnt2>0)&&(inportb(Port+0xE)<128))//直到 2XEH 端口第 7 位为 1
            --Cnt2;//延迟 100 毫秒
        if((Cnt2==0)||(inportb(Port+0xA)!=0xAA)) {//判断是否找到声卡端口
            --Cnt1;
            if(Cnt1==0) {//该基地址不对，复位失败
                Cnt1=NumberOfTimes1;
                Port=Port+0x10;
            } } else//复位成功
            Found=1; }
    if(!Found){
        printf("Nobase port found!\n");//最终没有找到声卡
        printf("\nDSP not reseted!\n");
        exit(0); } }
```

12.2.3　写 DSP

写 DSP 的过程，实际上就是播放声音文件。这里将使用到 2XCH 端口作为命令输出、数据输出和缓冲数据状态输入功能。以下给出播放 8 位单声道声音的具体步骤：

（1）向命令输出端口发送 10H 命令；

（2）等待缓冲清空，循环读取缓冲数据输入端口，直到第 7 位为 0；

（3）向数据输出端口输入采样数据；

（4）延迟，就是采样率除以时钟频率的时间周期。

稍候的 wav 文件播放例程将给出具体的程序实现办法。

12.3　播放 wav 文件

wav 文件是为 Windows 发明的音乐文件格式。这种文件是利用饮品处理和数字录音电路对实际声音信号进行数字化处理后所得的声音波形文件，其中包含所记录的声音实际波形的所有数据，只有保存这些数据才能在以后的重放时恢复所记录的声音音色、音调等。对声音信号波形的采样和数字化处理涉及“采样频率”和采样数据“位”两个术语。

下面给出了播放*.wav 文件的程序。在播放前首先需要在硬件端口 210H~260H 中查找是否安装了声霸卡。然后根据是否是立体声、采样精度的不同来提取声音样本，然后通过改写时钟中断并将数据输出到声卡数据输出端口来播放音乐。

12.3.1　WAV 文件格式

wav 文件分头文件区和数据区，头文件区一共 44 个字节。表 12-4 是 wav 文件头文件区重要参数。

表 12-4　wav 的头文件区参数

偏移（字节）	保存内容
0-1L	厂家 ID（通常：RI）
2-3L	产品 ID（通常：FF）
4L	驱动程序版本号（通常：1 或者 2 或者 r 或者空）
5L-17L	产品名称（例如：wavEfmt）
18-21L	所支持的标准格式
22-23L	通道数
24-25L	音频
34-35L	采样率
40-43L	数据区长度

通道数：单通道为 01H，00H；双通道为 02H，00H。

音频：11.025KHz 为 11H，2BH；22.05KHz 为 22H，56H；44.1KHz 为 44H，ACH。

采样率：8BIT 为 08H，00H；16BIT 为 10H，00H。

表 12-5 是每个采样数据所占字节数和通道数、采样率的关系。

表 12-5　wav 的头文件区

通道数	采样率	单位采样字节
单	8BIT	1
单	16BIT	2
双	8BIT	2
双	16BIT	4

表 12-6 是通道数、音频、采样率和每秒钟数据传送平均字节数的关系。

表 12-6　采样与数据传送的关系

通道数	音频（KHz）	采样率（字节）	每秒传送字节	播放时间（秒）
单	11.025	1（8bit）	11025	64
单	11.025	2（16bit）	22050	32
单	22.05	1	22050	32
单	22.05	2	44100	16
单	44.1	1	44100	16
单	44.1	2	88200	8
双	11.025	1	22050	32
双	11.025	2	44100	16
双	22.05	1	44100	16
双	22.05	2	88200	8
双	44.1	1	88200	8
双	44.1	2	176400	4

12.3.2　WAV 文件播放

在熟悉声卡 DSP 编程原理和 wav 文件结构后，我们提出 wav 文件播放方案：

（1）读取 wav 文件头信息；

（2）测试声卡 DSP 端口；

（3）修改定时器频率；

（4）改写时钟中断服务程序；

（5）读取 wav 文件数据区，并且写入 DSP，直到播放结束；

（6）恢复定时器频率和时钟中断服务程序。

其中第 3 和第 4 个步骤需要详细介绍一下。

第 3 个步骤中我们要修改定时器的频率，其目的是使得每次输出采样声音的时候能够按照音频要求进行正确时间的延迟。比如我们首先将定时器的触发中断频率设置为音频基速率 44100Hz，那么当 wav 文件采样在 22050Hz 音频情况下，对每个采样数据延迟 2 个定时器倒计数时间。1193180Hz 是定时器芯片时钟速率，将它除以基础音频（44100Hz）便可以得到定时器的计数初值。每次当计数结束后触发时钟中断。修改定时器频率的过程如下：

（1）将 1193180 除以音频 44100 得到初值；

（2）向定时器 43H 命令端口写入 36H；

（3）将初值的低字节写入 40H 端口；

（4）将初值的高字节写入 40H 端口。

第 4 个步骤实际是为了配合第 3 步一起实现采样声音正确延迟的。我们在中断服务程序中进行累加计数，从而知道现在定时器进入了第几次倒计数。同样，22050H 音频进行 2 次到计数，如何知道是 2 次就要看中断服务程序中的计数值。当计数值等于文件音频除基础音频（比如 44100/22050）的时候，计数变量就在中断服务程序中清零了。还要注意的一

点就是时钟中断服务程序中还必须调用旧的时钟中断服务程序。

还有一点值得指出的是，在设置和恢复中断的时候都必须关中断，直到完成后再开中断。播放 wav 文件的例程 wav.c 请查阅所附光盘的“source\12”目录。

12.4　WAV 背景音乐

wav 文件背景音乐的实现方法和扬声器背景音乐的实现原理基本类似，也是通过时间中断函数来实现，不过复杂了一些。同时 wav 背景音乐播放和 wav 文件播放也有所不同。以下给出相对前者的变化：

（1）我们必须将 wav 文件中的数据读到内存中，如果无法一次读出则必须依靠多次读取（类似于缓冲区）；

（2）所有播放任务都在新的时钟中断服务程序中进行。

（3）wav 中每个声音数据的延迟时间都是相等的，为 44100/wfhead.nsamps(44100/音频)。

具体的例程 wavgd.c 请查阅所附光盘的“source\12”目录。

12.5　本 章 小 结

游戏最多可以动用人的五觉中的三觉，倘若我们没有办法在游戏中触发听觉就无法实现真正优秀的游戏。C 语言提供了简单的扬声器相关函数 sound()和 nosound()，再加上 delay()函数我们就可以实现声音的播放了。

还可以通过对硬件端口进行操作来直接实现扬声器发声的功能。扬声器的输入寄存器分配的 I/O 端口为 60h 和 62h，输出寄存器控制端口为 61h。其中对于控制端口 61h 当其低 2 位如果同时是 1 表示允许发声，当位 1 为 0 则关闭扬声器电路而无法发声。在允许发声之前首先设定好发声的频率，这是针对 2 号定时器端口 43h 和 42h 进行的。要向 43h 端口写入 0b6h，初始化定时器的方式寄存器，此时定时器 2 等待接收计时常数。之后在 42h 端口写入一个常数（2 个字节），此常数实际就是 123280H（533H*896Hz）除以要产生的声音频率。

使用时间延迟函数来实现声音播放会导致游戏的暂停，我们必须考虑在背景中播放声音，这样游戏可以顺利地运行。扬声器背景音乐的实现通过对事件中断进行改写来实现。DOS 下的时间中断每秒钟产生 18.2 次，在时间中断的函数中加入：

（1）设置累加变量；

（2）调用 Sound(HZ[])函数,具体调用 HZ[]中哪个数组要根据累加数来决定；

（3）当累加数和每个音符要求延迟时间相同的时候调用 nosound()函数（此时累加数清零），实际上就是完成 delay()的工作。

扬声器发声事实上根本无法满足用户对游戏多媒体的要求，我们必须通过驱动声卡来实现真正的音乐在音像中播放。声卡主要包括 DSP 芯片、CODEC 芯片、控制芯片、FM 合成芯片和混音器芯片。DSP 芯片是用来对声音进行输入和输出的，所有的数模相互转换（A/D 或者 D/A）都在这个芯片中进行。

最流行的音乐问件格式是 wav 文件，我们可以通过 DSP 芯片播放 wav 文件。其播放方案如下：

（1）读取 wav 文件头信息；

（2）测试声卡 DSP 端口；

（3）修改定时器频率；

（4）改写时钟中断服务程序；

（5）读取 wav 文件数据区，并且写入 DSP，直到播放结束；

（6）恢复定时其频率和时钟中断服务程序。

至于 wav 文件背景音乐的播放和扬声器背景播放的原理类似，只是在实现过程中复杂了一些。我们必须将 wav 文件中的数据不断读取到内存中，然后再用经过改写过的时间中断函数来读取。

学后建议

（1）尝试制作带有背景音乐（包括扬声器背景音乐和 wav 声卡背景音乐）的游戏（如同俄罗斯方块中配有俄罗斯传统音乐）；

（2）研究 MIDI 文件的格式，以及声卡对应的编程端口，尝试用 C 语言实现对 MIDI 文件的声卡播放；

（3）研究声卡模拟转数字（A/D）相关的端口，制作 C 语言录制外界声音的程序。

第13章　内 存 技 术

本章导读

我们知道内存是游戏中十分关键的一个部分。事实上用 DOS 下 64K 常规内存做游戏是不可能的，于是将几十兆的扩充内存应用起来便成为非常重要的任务了。

DOS 下我们可以通过编程使用到的内存种类主要包括四种（部分之间互有包含关系）：

（1）常规内存：基本内存，64K，用 C 语言内存申请函数可以直接申请；

（2）扩展内存：64K 以后的内存，包括 UMB、HMA 和部分扩充内存；

（3）扩充内存：就是插在主板上的内存（16M，32M，64M，128M 爱多少有多少）；

（4）保留内存：384K 上位内存（Upper Memory Blocks,UMB），只有系统可以使用。

从 DOS 下内存的管理角度说，主要包括以下 3 种内存管理方式：

（1）扩展内存管理程序：himem.sys；

（2）扩充内存管理程序：在扩展内存管理程序基础上使用 emm386.exe；

（3）保护模式：DPMI，使程序运行在保护模式下，CPU 以虚拟地址方式工作，可以直接访问大内存。

在 Watcom C 和 Pascal 语言中提供了保护模式编程，这样可以直接申请大内存；而 TC 并没有提供保护模式编程方式。这便是为什么 TC 写游戏不被青睐的主要原因。不过当我们利用 DOS 保护模式接口将 DOS 运行在保护模式下以后，可以通过使用支持 BC3.0 的 DPMI 保护模式函数库来使用大内存。

本章将主要介绍常规内存调用、扩展内存技术和扩充内存技术。

本章重点

（1）扩展内存的管理方式　提供的编程端口，使用扩展内存的方法及其程序实现；

（2）扩充内存的管理方式　提供的编程端口，使用扩充内存的方法及其程序实现；

（3）运用内存技术实现中文字库的内存调入使用和实现游戏大地图的调入使用。

13.1　常 规 内 存

我们在游戏中经常用到常规内存，本节主要介绍常规内存的申请、使用和释放函数和指向内存的指针操作函数，详细函数（malloc(), farmalloc(), calloc(), farcalloc(), realloc()和farealloc()等）的介绍请查阅所附光盘的“book\附录 F”。

对于这些函数的使用方法这里不做详细的介绍了，下面给出一个简单的内存申请例程：

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>
void main(void) {
```

```
char far *fptr;
char *str = "Hello";
fptr = (char far *) farcalloc(10, sizeof(char));//动态申请远堆空间
//将 str 字符串中的内容复制到 fptr 中
movedata(FP_SEG(str), FP_OFF(str),FP_SEG(fptr), FP_OFF(fptr), strlen(str));
printf("Far string is: %Fs\n", fptr);
farfree(fptr); }//释放远堆内存
```

13.2　内 存 结 构

内存的结构表示如图 13-1。

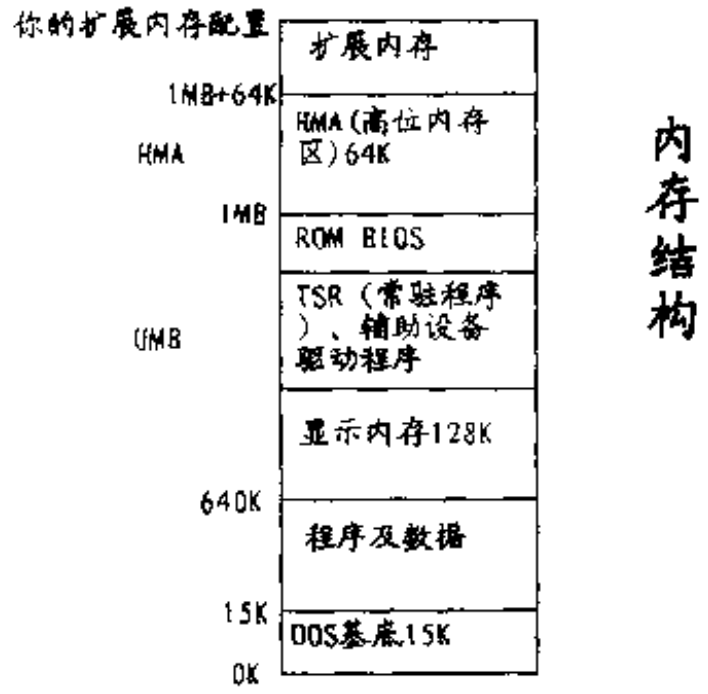


图 13-1　内存的结构示意

写游戏的时候常常会碰到无法向常规内存申请到足够的数量内存的情况，这时候我们只能考虑调用的扩充内存、扩展内存。要使用它们首先必须安装内存管理程序。DOS 为我们提供了两个内存管理程序 himem.sys 和 emm386.exe。

在 CONFIG.SYS 文件中我们可以根据自己的需要进行类似如下的设置：

```
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE /RAM
```

在设置好内存管理程序之后我们便可以通过 XMS（扩展内存规范）技术和 EMS（扩充内存规范）技术使用 64K 以上，电脑配置内存总量以下的大量空闲内存空间了。

C 语言使用大空间内存的方法比 Pascal 语言和 watcom C 的确复杂了很多，后者只需要进入保护模式就可以线性地使用大内存了，而前者使用大内存要用到下两节介绍的 XMS 技术和 EMS 技术。

13.3　XMS 技术

13.3.1　XMS 基本知识

XMS 技术是扩展内存规范（Extended Memory Standard）的简称。扩展内存是指 1M 寻址范围之外的内存。在 XMS 中扩展内存还包括高端存储区 HMA 和上位内存块 UMB。

UMB 是指在 DOS 内存 64KB 边界和 1M 边界之间的内存。HMA 则是指当 CPU 处于实模式并且第 21 条地址线（A20 线）被激活时，CPU 就可访问的一块 65520 字节的内存。要使用 XMS 内存，必须在 config.sys 中装入 XMS 驱动程序：

```
device=HIMEM.SYS
```

XMS 驱动程序提供了五组功能：驱动程序信息、HMA 管理、A20 线管理、扩展内存管理和 UMB 管理。另外的两个功能是检索 XMS 驱动程序是否存在和 XMS 驱动程序控制功能的地址。下面列出了 XMS 主要功能调用。

注意在调用 XMS 功能之前，必须先检查 XMS 驱动程序是否存在。如果存在，还必须获得 XMS 功能的远调用入口地址。

13.3.2　XMS 基本函数

XMS 和应用程序之间的接口是 INT 2FH,AH=43H。应用程序通过 INT 2FH,Ah=43H 获得 XMS 驱动程序控制功能的入口地址，并以子程序远调用的形式使用和管理由 XMS 定义的内存。XMS 功能调用索引表见附录 D。

13.3.3　XMS 调用基本程序

程序使用 XMS 的基本过程：

（1）调用 XMS 检测函数，判断 XMS 驱动程序是否安装：（如果不存在,config.sys 加载 himem.sys，重新启动）

```
char test_xms();
```

（2）调用取 XMS 驱动器入口地址函数：（稍后的 XMS 操作函数都要使用到这里取到的指向 XMS 驱动器程序的指针）

```
void get_driver_address();
void (far *xms)(void);
```

（3）调用检测 XMS 版本函数：（可省略）

```
void get_xms_version(unsigned &xms_ver,unsigned &int_ver);
```

（4）调用检测 XMS 可用空间函数，如果空间过小（<64KB）就没有什么实用价值了：（可省略）

```
char query_free_xms(unsigned &max_block,unsigned &total);
```

以下完成写入到 XMS 扩展内存的工作：

（5）将要放入 XMS 的内容准备好，并告知需要的 XMS 扩展内存空间；

（6）调用分配扩充内存函数申请要使用到的 XMS 空间：

```
char allocate_xms(unsigned size,unsigned &handle);
```

（7）对移动扩展内存块结构内容赋值，包括源地址（刚才准备好的内容）和目的句柄（刚才申请的 XMS 句柄）：

```
xmm.byte_count=字节数;
xmm.source_handle=0;
xmm.source_offset=源地址;
xmm.destination_handle=handle;
xmm.destination_offset=0;
```

（8）调用移动扩展内存函数：

```
char move_xms(xms_mov &xmm)
```

以下完成从 XMS 扩展内存读出的工作：

（9）对移动扩展内存块结构内容再次赋值，包括源句柄（刚才存入内容到 XMS 的句柄）和目的地址（为当前的使用而准备的常规内存地址）：

```
xmm.byte_count=字节数;
xmm.source_handle=handle;
xmm.source_offset=0;
xmm.destination_handle=0;
xmm.destination_offset=目的地址;
```

（10）调用释放 XMS 指定句柄的函数。

```
char free_xms(unsigned handle);
```

程序 xms.c 请查阅所附光盘的“source\13”目录。

13.3.4　将中文字库调入 XMS

在第 7 章中文显示中我们已经介绍过如何通过字库文件调用来实现中文的显示。然而，在读取每个中文字的时候都进行了打开和关闭字库的操作，这样的效率是可以想而知的。对于西文字库，由于它的 ASCII 码一共只有 256 个，于是我们可以采取将所有的西文字库一次性读取到数组中。比如 ASC16 字库的大小只有 4096 个字节。再来看看 hzk16 中文字库文件其大小是 267161 字节，我们很难在常规内存中申请到如此大的空间（有点羡慕 Pascal 语言的保护模式了）。于是在第 7 章我们放弃了这个思路，现在我们学会了用 XMS 来申请更多的扩展内存，那么我们就来享受一下将一个中文字库全部调入内存，然后从扩展内存中读取我们所要的中文字符点阵。

1. 字库调入 XMS

将字库调入 XMS 实际上就是从文件先读取数据到常规内存然后再将常规内存中的数

据复制到申请的扩展内存中。然而与前一个 XMS 测试程序不同的是前面我们将 Autoexec.bat 调入扩展内存只需要申请 1K 空间就足够了，而现在必须申请 300K 扩展内存。同时由于前面 1K 内存在常规内存中也很容易申请到，于是可以一次性从文件读到常规内存再读到扩展内存。而现在的这个 hzk16 字库文件差不多有 260K 左右，我们只能拿出曹冲称象的智慧和愚公移山的精神来完成它。具体的做法就是，每次从文件读取 4K 的数据到常规内存，然后再从常规内存复制到扩展内存，这样不断循环，直到所有的数据都放入扩展内存。以下给出具体的函数：

```
char Load_Hzk16_Xms(void){
  int e,hzk;
  char flag;
  unsigned a,pageaddr,hzhandle;
  unsigned char far *tmp,buffer[4096];
  unsigned max_block,total;
  hzk=open("hzk16",O_RDONLY|O_BINARY);//打开字库
  if (hzk!=-1) {//判断打开是否成功
if (test_xms()) {//测试 XMS 是否存在
  get_driver_address();//取得 XMS 入口地址
  query_free_xms(&max_block,&total);//查询空间
  if (max_block>300) {//判断是否大于 300K
   if (!allocate_xms(300,&hzhandle)) {//申请 300K 扩展内存空间
     unsigned long s=0;
     xmove.byte_count=4096;//设置每次复制字节数量
     xmove.source_handle=0;
     xmove.source_offset=(unsigned long)buffer;//设置数据源为每次从文件
    //读取 4096 字节数据所存放的 buffer 数组
     xmove.destination_handle=hzhandle;//设置目的内存的句柄
     for (;e>0;) {//只要文件没有结束
    e=read(hzk,buffer,4096);//每次从文件读取 4096 字节的数据
    xmove.destination_offset=s;//设置目的地址每次的偏移
    s+=e;//扩展内存中的偏移每次增加 4096
    move_xms(&xmove);//进行内存间复制
     } close(hzk);//关闭文件
     xmove.byte_count=32;//设置每次读取字节数
     xmove.source_handle=hzhandle;//设置源内存地址为刚才申请的扩展内存句柄
     xmove.destination_handle=0;
     return(0);
  } } } }
return(1); }
```

2. 从 XMS 读取一个中文字

由于前一个函数已经设定了源内存地址的句柄就是刚才读入字库文件的扩展内存句柄，同时设定了每次读取的数量为 32 个字节（1 个 16*16 中文字的点阵）。所以在读取中文字的函数中，我们只需要设定源内存地址中的偏移和目的地址就可以了。而源内存地址的偏移实际上就是我们原先的中文字在字库文件中的偏移，它可以通过中文字的两个扩展 ASCII 码读取而来。而目的地址实际上就是存放 32 个字节点阵的数组。以下给出具体函数：

```
void Read_Hzk16_Xms(int ch0,int ch1,unsigned char *buffer){
  unsigned p1,p2;
  p1=ch0-161;
  p1=p1*94+ch1-161;//计算中文字在字库内（申请的扩展内存中）的字偏移
   xmove.source offset=p1;
   xmove.source_offset*=32;//设置源偏移（中文字在字库内的偏移）
   xmove.destination_offset=(unsigned long)buffer;//设置目的地址为 32 个字节的字符数组
   move_xms(&xmove); }//进行内存间复制
```

当我们完成从扩展内存中读取一个中文字 32 个字节的点阵到数组后，以后的文字显示工作就和原先的完全一样。具体的程序代码 hzkxms.c 请查阅所附光盘的“source\14”目录。

13.4　EMS 技术

13.4.1　EMS 基本知识

扩充内存规范(Expaned Memory Specification)EMS 从功能上定义开关控制存储器的内存扩充子系统规则，它有硬件扩充模块和相应于这些模块的驻留驱动程序构成。应用程序可以以 16KB 的页为单位使用扩充内存，这些页面映像到称为页框 (Page Frame) 的连续的 64KB 区域中，页框位于 DOS 使用的常规内存（0~64KB）上方。页框的精确位置可由用户确定，但必须保证和其它硬件不发生冲突。

EMS 技术是扩充内存规范（Expanded Memory Standard）的简称，是为了解决 8086 和 8088 计算机内存无法扩充的问题而提出的。这一技术是在 384K 保留地址空间中用 64KB 的地址来访问较大的几兆的内存空间。它通过一对多的映射技术来实现使用大内存空间的。

常规内存中有 64KB 连续的内存空间，被称为页框 (Page Frame)，这 64KB 空间被又分为 4 个 16KB 的页。通过这里的 4 个页来和扩充内存进行数据交换。

当要将数据从扩展内存中取出的时候，每次将需要的扩充内存页面号码和常规内存的映射页框号码对应起来，然后调用映射函数，处理后常规内存中映射页框内的数据就是扩充内存的数据了。对于大量扩充内存中的数据值需要循环往复的操作就能够将所有扩充内存中的内容取出。

当要向扩展内存写入数据的时候，先将扩展内存中空闲的内存的页码和常规内存中映射页框号码进行映射，然后将数据放入参加映射的页框中，然后数据就直接进入扩展内存了。如果你要放入扩展内存的数据大于 64KB，可以通过几次写入页框并对应扩展内存不同

空间进行映射。实现原理如图 13-2 所示。

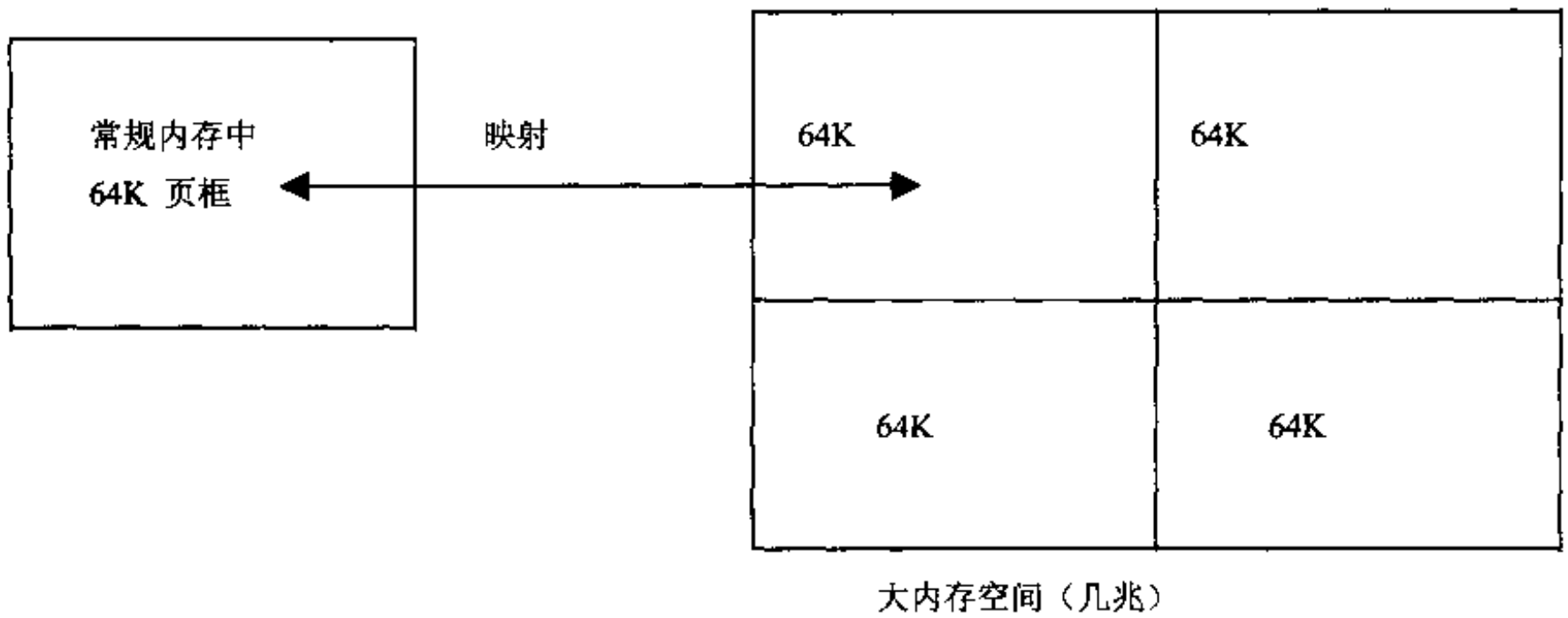


图 13-2　常规内存与扩展内存映射原理示意

要使用 EMS 的时候需要在 config.sys 中加载：

```
device=himem.sys
device= emm386.exe /ram
```

在 autoexec.bat 中加载：emm386.exe

应用程序借助软中断 int 67H 直接与 EMM 通讯。EMS 功能调用索引函数请查阅附录 D。

13.4.2　EMS 调用基本程序

程序使用 EMS 的基本过程。以下完成准备工作：

（1）调用 EMS 检测函数，判断 EMS 管理程序是否安装；（如果不存在,config.sys 写入 device=himem.sys 以及 device=emm386.exe /ram，autoexec.bat 写入 emm386，重新启动；）

```
char test_ems()
```

（2）调用 EMS 状态检测函数，判断 EMS 是否激活；

```
char get_EMS_status()
```

（3）调用 EMS 版本测试函数；（可省略）

```
char get_EMM_version(unsigned char *version)
```

（4）检测 EMS 页面总数和未使用页面数；

```
char get_number_of_pages(unsigned *avail,unsigned *total)
```

（5）取得常规内存中 64k 映射地址；（通常是 d0000H,可以不检测而直接设置，有的笔记本为 e0000H）

```
char get_page_frame_segment(unsigned *segment)
```

```
unsigned char far *segment_buffer=(char far *)0xD0000000L;
```

（6）取得可用句柄数量；

```
char get_number_of_EMM_handles(unsigned *handle_numbers)
```

将数据写入 EMS 扩充内存：

（7）确定要申请 EMS 扩充内存大小；(大小为 n*16k)

```
page_numbers=n;
```

（8）申请 EMS 扩充内存；

```
char allocate_memory(unsigned *handle,unsigned page_numbers)
```

（9）验证当前申请到的内存大小；（可省略）

```
char get_pages_owned_by_handle(unsigned handle,unsigned *page_numbers)
```

（10）确定当前要映射的物理地址页（0~3 中选择一页）和逻辑地址页（0~n-1 中选择一页）；

```
physical_page=M;
logical_page=N;
```

（11）将需要的输入的数据写入常规内存中 EMS 映射部分 4 页中；

（12）常规内存中 EMS 映射部分和 EMS 扩充内存进行映射；（一页一映射）

```
char map_memory(char physical_page,unsigned logical_page,unsigned handle)
```

从 EMS 扩充内存读出数据：

（13）将 EMS 扩充内存中刚才保存的内容映射到常规内存中 EMS 映射部分中；

```
char map_memory(char physical_page,unsigned logical_page,unsigned handle)
```

（14）将常规内存中 EMS 映射部分中数据复制到需要的其他常规内存位置（比如显存）；

程序 ems.c 请查阅所附光盘的“source\16”目录。

程序运行显示结果：

```
abcdefghijkl
a
abcdefghijkl
```

这三行显示结果都是字符串 buffer 的内容显示：

（1）是给 buffer 赋值 abcdefghijkl 后的结果；

（2）是将 buffer 复制给常规内存映射页框（同时给扩充内存后）赋值 a 后的结果；

（3）是将扩充内存和常规内存映射后，从常规内存物理页框取到 buffer 内的数据。

13.4.3　将中文字库调入 EMS

在前面我们尝试将一个中文字库全部调入扩展内存，然后从 XMS 中读取我们所要的中文字符点阵。现在我们同样可以尝试使用扩充内存（EMS 技术）来存放这个字库。

1. 字库调入 EMS

将中文字库调入 EMS 的思路和调入 XMS 中是相似的，只是多了一个步骤。实际上就是从文件先读取数据到数组变量然后再将字库数组变量中的数据读取到常规内存中第一个物理映射页，最后将物理映射页数据映射到申请的扩充内存的对应逻辑页中。具体的做法就是，每次从文件读取 4K 的数据到数组变量，然后再将字库数组变量数据按字节传到常规内存第一个映射页，完成后再循环读取文件，4 次以后第一个映射页满（16K），此时从常规内存第一个映射页将字库数据映射到申请的扩充内存对应页，这样不断循环，直到所有的数据都放入扩充内存。以下给出具体的函数：

```
char Load_Hzk16_Ems(void){
  int e,hzk,page,i,j;
  char flag;
  unsigned a,pageaddr;
  unsigned char far *tmp,buffer[4096];
  unsigned max_block,total;
  unsigned totalpage;
  hzk=open("hzk16",O_RDONLY|O_BINARY);//打开字库文件
  if (hzk!=-1) {//判断字库是否能够打开
if (test_ems()) {//判断 EMS 是否存在
  if (!get_EMS_status()) {//取得 EMS 状态
    flag=get_number_of_pages(&a,&totalpage);//取得 EMS 自前页面数量
    page=28;//将要申请的页面数量
    if (a>=page) {//判断空前页面是否够
      flag=get_page_frame_segment(&pageaddr); }//取得映射地址，通常是 D000H，笔记本
      //常常是 E0000H
if ((!flag) && (!allocate_memory(&hzhandle,page))) {//申请扩充内存页面
  fc=(char far *)MK_FP(pageaddr,0);
  tmp=fc; a=0;
  for (e=1;e!=0;) {//不断映射直到字库文件结束
if (tmp==fc+16384) {//判断字库数据是否装满了第一个物理映射页（16K）
 //从影像页存入扩充内存
  map_memory(0,a,hzhandle);
  a++; tmp=fc;//扩充内存逻辑页面加 1，常规内存第一个物理映射页重新开始存放字库数据
} e=read(hzk,buffer,4096);//将文件数据读入字库数组
for (i=0;i<e;i+=32) {
```

```
    for (j=0;j<32;j++) *tmp++=buffer[i+j];//将字库数组数据读入常规内存中的物理映射页
     //第一页，同时进入扩充内存
  } }
    while (tmp-fc<16384) *tmp++=0;//最后 一次影像前对不满 16K 的字库数据补 0
    close(hzk);//关闭文件
 return(0); } } }
} return(1); }
```

2. 从 EMS 读取一个中文字

相对于 XMS 可以直接从扩展内存中取得任意指定字节数的数据而言，EMS 使用的映射方法至少要一次读取 16K 数据。也就是说即使我们只要一个中文字的 32 个点阵字节也仍然需要读取 16K 的数据。这时我们无法通过直接计算中文字在字库中的字节偏移（在扩充内存中的偏移）来读取它。取而代之的方法是我们需要根据中文字在字库中的偏移换算出此中文字所在的扩充内存逻辑页面号码，然后将这个页面读取到常规内存第一个映射页面中，之后再对这个页面取得中文字的页内（16K 以下字节）偏移，最后从这个地址开始读取 32 个字节的数据。以下给出具体函数：

```
void Read_Hzk16_Ems(int ch0,int ch1,unsigned char *buffer){
  int i;
  unsigned p1,p2;
  unsigned char far *dot;
  p1=ch0-161;
  p1=p1*94+ch1-161;//计算当前中文字在字库中的字偏移
  p2=p1>>9;//将字偏移乘以 256，再乘上每个字的字节偏移 32，p2 正好是
  //中文字在字库中的字节偏移的 16K 倍，p2 为当前字所在的页偏移
  map_memory(0,p2,hzhandle);//将扩充内存中第 p2+1 个逻辑页面映射到常规内
  //存第一个物理映射页面
  p2=(p1 & 511)<<5;//将字偏移 p1 对 0.5k 求余然后乘以 32 个字节，实际上就
  //是使 p2 指向 16K 映射页面内此中文字的字节偏移
  dot=fc+p2;//取得此中文字在常规内存中的地址
  for(i=0;i<32;i++) buffer[i]=dot[i]; }//将 32 个点阵字节复制给 buffer 数组
```

当我们完成从扩充内存中读取一个中文字 32 字节的点阵到数组后，以后的文字显示工作就和 XMS 的实现办法完全一样了。具体的程序代码 hzkems.c 请查阅所附光盘的“source\14”目录。

13.4.4　全方位拉屏

在第 9 章中已经介绍了拉屏技术，并且给出了横向拉屏的例程。但是由于全方位拉屏需要极大的内存空间，所以该范例无法在常规内存中实现全方位拉屏。在学习了本章扩展内存技术和扩充内存技术后，完全可以获得所需大小的内存空间（除非你的电脑只配置了

很少的内存，不过 16M 已经足够了），这样一来实现全方位拉屏就不再是纸上谈兵了。

这里想再简单地介绍一下全方位拉屏技术。我们经常在 RPG 游戏中了解到全方位拉屏技术的应用，比如“仙剑”中主人公可以在地图中以 45 度方向前进，以及 C&C 中坦克等对象在地图中任意方向移动。全方位拉屏首先需要将整个或者局部的地图（远远大于屏幕大小）放入内存，然后不断根据用户输入的信息进行计算获得当前地图所在的视区。全方位拉屏技术实现上最大的困难在于以下两点：

（1）大容量的内存申请；

（2）在视区变化时内存地址的计算。

这里我们尝试实现一个全方位拉屏的例子：首先地图大小是屏幕的 16 倍，也就是由 16 张 320*200 的 pcx 文件组成（横方向和纵方向为 4 行 4 列）。屏幕中有一个主人公子画面可以按照东南西北四个方向在地图中行走（主人公始终在屏幕中央，实际移动的是地图），最后主人公可以进入地图中的某些特定场所。

以下给出程序实现主要过程：

（1）将地图全部调入 EMS 中；

（2）将当前视区显示于屏幕；

（3）接受键盘用户控制；

（4）保存当前视区到 EMS 中；

（5）判断是否进入特定场所，如果是进入特定场所的函数，否则继续；

（6）根据移动方向将保存当前视区的 EMS 内仍然在新视区中的画面读入屏幕对应位置；

（7）从 EMS 中读取新补充入屏幕视区的画面；

（8）循环第 3~7 步骤，直到退出。

由于此程序长度为 3100 多行，所以在书中不给出完全的代码，仅给出主要的程序实现思路分析和部分函数代码。

1. 地图调入 EMS

在程序编制前，我首先画了一张 1280*800 像素 256 色的上海地图，然后将它切割成 16 张 320*200 的画面，最后保存为 pcx 文件。

考虑将这 16 张地图文件保存入 EMS，我们发现每个 pcx 文件转化成 320*200 像素的 256 色位图后正好是 64000 个字节，完全可以用 4 个页面来保存。于是我们决定为 16 个地图文件申请 16 个扩充内存句柄，而每个句柄 4 个页，这样共计需要 64 个扩充内存的页。

以下是将地图读入扩充内存的代码段：

```
unsigned handles[MAP_NUM+1];//地图中每个 320*200 图像的句柄（一共 16 个，每个 64K、4 个页）
unsigned handle;//当前句柄
char **file;//地图文件名称数组
file[0]="dxfy1.pcx";//载入地图文件名
file[1]="dxfy2.pcx";
file[2]="dxfy3.pcx";
```

```
file[3]="dxfy4.pcx";
file[4]="dxfy5.pcx";
file[5]="dxfy6.pcx";
file[6]="dxfy7.pcx",
file[7]="dxfy8.pcx";
file[8]="dxfy9.pcx";
file[9]="dxfy10.pcx";
file[10]="dxfy11.pcx";
file[11]="dxfy12.pcx";
file[12]="dxfy13.pcx";
file[13]="dxfy14.pcx";
file[14]="dxfy15.pcx";
file[15]="dxfy16.pcx";
...
test_ems_active();//测试 EMS 是否激活
...//省略了判断 EMS 页面是否大于 64（16 个文件，每个文件转化为位图后正好是 4 个页面）的过
程
for(i=0;i<=15;i++) {
handle=save_ems_file(file[i]);//将 pcx 文件保存到 EMS 中
handles[i]=handle; }
...
```

2. pcx 文件调入 EMS

将地图调入 EMS，实际上是将许多 pcx 文件调入 EMS 的过程。将一个 pcx 文件读入 EMS 的具体过程如下：

（1）一次申请 4 个页（64K）；

（2）将扩充内存中 4 个逻辑页和常规内存的 4 个物理映射页面进行映射；

（3）将 pcx 文件读入常规内存 4 个物理映射页面（存入扩充内存）。

以下给出将 320*200 像素 256 色 pcx 文件调入 EMS 中的函数：

```
unsigned save_ems_file(char *filename){
char result;
int i;
unsigned page_numbers,handle;
unsigned segment;
unsigned logical_page,physical_page;
int result_num;
unsigned handle_numbers;
FILE *fp;
```

```
PCX_picture background_PCX;
page_numbers=4;//申请页面数为 4
result=allocate_memory(&handle,page_numbers);//申请 64K 扩充内存
for(i=0;i<4;i++){
physical_page=i;//设定当前常规内存物理映射页面号
logical_page=i;//设定当前申请到的扩充内存逻辑页面号
result=map_memory(physical_page,logical_page,handle); }//将常规内存页面和扩充内存页面
映射
PCX_Load_Ems(filename,(PCX_picture_ptr)&background_PCX,1);
//将 pcx 文件读取到常规内存 4 个物理映射页面，实际读入对应的 4 个扩充内存页
return(handle); }
```

PCX_Load_Ems()函数实际上就是在 PCX_Load()函数基础上进行修改的。它将 pcx 文件读取的点阵送入常规内存中映射页面的对应地址（替代了 PCX_Load()函数申请的内存空间）。以下给出此函数：

```
void PCX_Load_Ems(char *filename, PCX_picture_ptr image,int enable_palette){
FILE *fp;
int num_bytes,index;
unsigned int count;
unsigned char data;
char far *temp_buffer;
fp = fopen(filename,"rb");
temp_buffer = (char far *)image;
for (index=0; index<128; index++){temp_buffer[index] = getc(fp);}
count=0;
while(count<=(unsigned int)SCREEN_WIDTH1 * SCREEN_HEIGHT3) {
    data = getc(fp);
 if (data>=192 && data<=255) {
   num_bytes = data-192;
    data  = getc(fp);
    while(num_bytes-->0) {
  segment_buffer[count++]=data;//读取的图像点放入映射页面
    } } else {
     segment_buffer[count++]=data;//读取的图像点放入映射页面
   } } fseek(fp,-768L,SEEK_END);
for (index=0; index<256; index++){
image->palette[index].red   = (getc(fp) >> 2);
image->palette[index].green = (getc(fp) >> 2);
image->palette[index].blue  = (getc(fp) >> 2);
```

```
} fclose(fp);
if (enable_palette) {
   for (index=0; index<256; index++) {
 Set_Palette_Register(index,(RGB_color_ptr)&image->palette[index]);
  } } }
```

指针变量 segment_buffer 的具体定义如下：

```
unsigned char far *segment_buffer=(char far *)0xD0000000L;
```

值得一提的是，通常在计算机中不需要检测 EMS 映射地址，而直接可以知道其地址是 D0000H，而在许多笔记本内为 E0000H。

3. 当前地图移动

最初的地图虽然可以直接从扩充内存的 4 个页调入映射页，然后复制到显存。但是，由于在 RPG 游戏中（比如“仙剑奇侠传”）主人公每次向一个方向走的时候屏幕只会向一个方向（东、南、西、北或者东南...）移动很少的距离（最多也就是几十个像素），同时在本程序中主人公每次在东西方向上移动单位为 16 个像素，在南北方向上移动单位是 10 个像素。于是我们将面临许多复杂情况和大量的页内地址计算工作。

地图向上移动的具体程序过程如下：

（1）取得保存当前屏幕的句柄；

（2）将常规内存 4 个物理页和此句柄进行映射；

（3）将当前屏幕内容复制到常规内存 4 个物理页所在地址；

（4）将 4 个物理页所在地址根据向上移动所发生的偏移复制还给屏幕对应位置；

（5）再将屏幕下部空缺的 20 行通过读取 EMS 对应地图页面补充进来。

以下给出一个向上方向移动时的函数：

```
void scr_up(unsigned handle){
int i;
unsigned logical_page,physical_page;
char result;
for(i=1;i<=5;i++){
change=-change;
Erase_Sprite((sprite_ptr)&baby);
save_ems_scr(handle);//将当前屏幕保存到 EMS
//将保存到扩充内存的当前屏幕（目前在常规内存物理映射页面中），向显存上移
asm push ds;
asm les di,video_buffer_up;
asm lds si,segment_buffer_up;
asm     mov cx,SCREEN_HEIGHT1*SCREEN_WIDTH1/2-SCREEN_UP*160;
//将 EMS 内容上移后的地址偏移复制给屏幕
```

```
asm     cld;
asmrep movsw;
asmpop ds;
pat_scr_up_all(i,x_times,y_times);//补充入新的地图部分
baby.curr_frame=baby_direction+2+2*change;
Behind_Sprite((sprite_ptr)&baby);
Draw_Sprite((sprite_ptr)&baby);
Delay(TIME);
} }
```

函数中向上移动时常规内存读入屏幕的首地址实际上要向下偏移 10 行，而读入图像的显存还是从左上角地址开始，并且复制的内存数量为屏幕大小减去 10 行的字节数，以下是它们的地址情况：

```
unsigned char far *segment_buffer_up=(char far *)0xD0000C80L;//偏移为 10*320
unsigned char far *video_buffer_up=(char far *)0xA0000000L;
```

这里给出其它移动方向上的地址情况：

```
unsigned char far *segment_buffer_down=(char far *)0xD0000000L;
//向下移动时常规内存起始地址无需偏移
unsigned char far *segment_buffer_left=(char far *)0xD000000AL;//左移偏移为 16 个象素
unsigned char far *segment_buffer_right=(char far *)0xD0000000L;
unsigned char far *video_buffer_down=(char far *)0xA0000C80L;
unsigned char far *video_buffer_left=(char far *)0xA0000000L;
unsigned char far *video_buffer_right=(char far *)0xA000000AL;
```

由于在东西（左右）方向移动时的情况和南北方向移动情况有所不同，最重要的是东西方向移动时内存地址复制不是连贯的。所以这里给出东西方向移动时的函数：

```
void scr_left(unsigned handle){
int i,j;
unsigned logical_page,physical_page;
char result;
i=0x0;
for(j=1;j<=4;j++){
change=-change;
Erase_Sprite((sprite_ptr)&baby);
//将当前显存内容保存入扩充内存，同时也映射了常规内存中的物理映射页面
save_ems_scr(handle);
for(i=0x1;i<=0xC8;i++) {//循环 200 行
//将保存到扩充内存的当前屏幕（目前在常规内存物理映射页面中），向显存左移
```

```
asm push ds;
asm les di,video_buffer_left;
asm lds si,segment_buffer_left;
asm     mov cx,SCREEN_WIDTH1/2-SCREEN_LEFT/2;//每次复制 304 个字节
asm     cld;
asm rep movsw;
asm pop ds;
video_buffer_left=(char far *)(0xA0000000L+0x00000140L*i);//显存左移地址进行偏移
segment_buffer_left=(char far *)(0xD000000AL+0x00000140L*i); }
//常规内存物理页面左移地址进行偏移
video_buffer_left=(char far *)0xA0000000L;
segment_buffer_left=(char far *)0xD000000AL;
pat_scr_left_all(j,x_times,y_times);//将新入视区的画面从 EMS 补入屏幕对应位置
baby.curr_frame=baby_direction+2+2*change;
Behind_Sprite((sprite_ptr)&baby);
Draw_Sprite((sprite_ptr)&baby);
Delay(TIME);
} }
```

以下是保存当前屏幕到 EMS 的函数：

```
void save_ems_scr(unsigned handle){
int i;
unsigned logical_page,physical_page;
//进行映射，保存入扩充内存
for(i=0;i<4;i++){
physical_page=i;
logical_page=i;
map_memory(physical_page,logical_page,handle); }
//将显存内容写入常规内存中物理映射页面，同时进入保存当前屏幕页面的扩充内存
asm push ds;
asm les di,segment_buffer;
asm lds si,video_buffer;
asm     mov cx,SCREEN_HEIGHT1*SCREEN_WIDTH1/2;
asm     cld;
asm rep movsw;
asm pop ds;
return(handle); }
```

4. 补充入屏幕的画面

补充入屏幕的新画面实际上就是从保存在 EMS 对应页的地图内读取的画面。我们在读取时可能会碰到几种情况：

（1）南北方向上需要读取一个 EMS 页（主人公在东西方向上走的次数为 5 的倍数时）；

（2）南北方向上需要读取两个 EMS 页（主人公在东西方向上走的次数不是 5 的倍数时）；

（3）东西方向上需要读取四个 EMS 页（主人公在南北方向上走的次数为 5 的倍数时）；

（4）东西方向上需要读取五个 EMS 页；

具体的计算相当复杂，如果有兴趣可以自行分析。以下是向上补充入屏幕函数：

```
pat_scr_up_all(int i,int x_times,int y times){
unsigned logical_page,physical page;
unsigned handle;
char result;
int k;
unsigned many;
if(x_times==0) {//x_times 为 0 表示在第一种情况，也就是只需要涉及一个页
for(k=0;k<4;k++){
physical_page=k;
logical_page=k;
handle=handles[j+4];
map_memory(physical_page,logical_page,handle); }//映射，从扩充内存读到常规内存页框
//将常规内存物理映射页框内容（刚才从扩充内存获得的）写入显存
segment_buffer_pat_up=(char far *)(0xD0000000L+0x00000C80L*(i-1)+
0x00003E80L*y_times);
video_buffer_pat_up=(char far *)0xA000ED80L;
asmpush ds;
asmles di,video_buffer_pat_up;
asmlds si,segment_buffer_pat_up;
asm     mov cx,SCREEN_WIDTH1*SCREEN_UP/2;
asm     cld;
asmrep movsw;
asmpop ds;
} else {//x_times 不是 0 表示在第二种情况，也就是只需要涉及两个页
for(k=0;k<4;k++){
physical_page=k;
logical_page=k;
handle=handles[j+3];
map_memory(physical_page,logical_page,handle); }
```

```
segment_buffer_pat_up=(char far *)(0xD0000000L+0x00000C80L*(i-1)
+0x00003E80L*y_times+0x00000028L*(8-x_times));
video_buffer_pat_up=(char far *)0xA000ED80L;
for(k=1;k<=10;k++){
many=20*x_times;
asmpush ds;
asmles di,video_buffer pat_up;
asmlds si,segment_buffer_pat_up;
asm     mov cx,many;
asm     cld;
asmrep movsw;
asmpop ds;
segment_buffer_pat_up=(char far *)(0xD0000000L+0x00000C80L*(i-1)
+0x00003E80L*y_times+0x00000028L*(8-x_times)+0x00000140L*k);
video_buffer_pat_up=(char far *)(0xA000ED80L
+0x00000140L*k); }
for(k=0;k<4;k++){
physical_page=k;
logical_page=k;
handle=handles[j+4];
map_memory(physical_page,logical_page,handle); }
segment_buffer_pat_up=(char far *)(0xD0000000L+0x00000C80L*(i-1)
+0x00003E80L*y_times);
video_buffer_pat_up=(char far *)(0xA000ED80L+0x00000028L*x_times);
for(k=1;k<=10;k++){
many=20*(8-x_times);
asmpush ds;
asmles di,video_buffer_pat_up;
asmlds si,segment_buffer_pat_up;
asm     mov cx,many;
asm     cld;
asmrep movsw;
asmpop ds;
segment_buffer_pat_up=(char far *)(0xD0000000L+0x00000C80L*(i-1)
+0x00003E80L*y_times+0x00000140L*k);
video_buffer_pat_up=(char far *)(0xA000ED80L+0x00000140L*k+0x00000028L*x_times);
} } }
```

以下是向左移动时补充入屏幕的函数：

```
pat_scr_left_all(int k,int x_times,int y_times){
int i;
unsigned logical_page,physical_page;
char result;
unsigned handle;
if(y_times==0) {//y_time 为 0 表示在第三种情况，也就是只需要涉及四个页
for(i=0;i<4;i++){
physical_page=i;
logical_page=i;
handle=handles[j];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0000136L;
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0xC8;i++){
asmpush ds;
asm les di,video_buffer_pat_left;
asmlds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asmrep movsw;
asmpop ds;
video_buffer_pat_left=(char far *)(0xA0000136L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
} }
else if(y_times==1) {//y_time 不是 0 表示在第四种情况，也就是需要涉及五个页
for(i=0;i<4;i++){
physical_page=i;
logical_page=i;
handle=handles[j];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0000136L;
segment_buffer_pat_left=
(char far *)(0xD0003E80L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x96;i++){
asmpush ds;
asm les di,video_buffer_pat_left;
```

```
asm lds si,segment_buffer_pat left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asm rep movsw;
asm pop ds;
video_buffer_pat_left=(char far *)(0xA0000136L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD0003E80L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x times)); }
for(i=0;i<=4;i++){
physical_page=i;
logical_page=i;
handle=handles[j+4];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA000BCB6L;
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x32;i++){
asm push ds;
asm les di,video_buffer_pat_left;
asm lds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asm rep movsw;
asm pop ds;
video_buffer_pat_left=(char far *)(0xA000BCB6L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
} } else if(y_times==2) {
for(i=0;i<4;i++) {
physical_page=i;
logical_page=i;
handle=handles[j];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0000136L;
segment_buffer_pat_left=
(char far *)(0xD0007D00L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x64;i++){
asm push ds;
```

```
asm les di,video_buffer_pat_left;
asmlds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asmrep movsw;
asmpop ds;
video_buffer_pat_left=(char far *)(0xA0000136L+0x00000140L*i);
segment_buffer_pat_left-
(char far *)(0xD0007D00L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x_times)); }
for(i=0;i<4;i++){
physical_page=i,
logical_page=i;
handle=handles[j+4];
map_memory(physical page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0007E36L;
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x64;i++){
asmpush ds;
asm les di,video_buffer_pat left;
asmlds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asmrep movsw;
asmpop ds;
video_buffer_pat_left=(char far *)(0xA0007E36L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
} } else if(y_times==3) {
for(i=0;i<4;i++){
physical_page=i;
logical_page=i;
handle=handles[j];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0000136L;
segment_buffer_pat_left=
(char far *)(0xD000BB80L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x32;i++){
```

```
asmpush ds;
asm les di,video_buffer_pat_left;
asmlds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asmrep movsw;
asmpop ds;
video_buffer_pat_left=(char far *)(0xA0000136L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD000BB80L+0x00000140L*i-0x0000000AL*(k-1)+0x00000028L*(7-x_times)); }
for(i=0;i<4;i++){
physical_page=i;
logical_page=i;
handle=handles[j+4];
map_memory(physical_page,logical_page,handle); }
video_buffer_pat_left=(char far *)0xA0003FB6L;
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
for(i=0x1;i<=0x96;i++){
asmpush ds;
asm les di,video_buffer_pat_left;
asmlds si,segment_buffer_pat_left;
asm     mov cx,SCREEN_LEFT/2;
asm     cld;
asmrep movsw;
asmpop ds;
video_buffer_pat_left=(char far *)(0xA0003FB6L+0x00000140L*i);
segment_buffer_pat_left=
(char far *)(0xD0000000L+0x00000140L*i+0x0000000AL*(k-1)+0x00000028L*(7-x_times));
} } }
```

13.5　本 章 小 结

游戏制作中声卡技术和内存技术是两大难点，内存是直接限制游戏制作质量的要素。DOS 下内存种类包括常规内存、扩展内存、扩充内存和保留内存 UMB。

学后建议

（1）尝试制作将 wav 声音文件一次性调入扩展内存，并且进行播放的程序；

（2）研究保护模式（DPMI）的工作方式、及在保护模式下内存访问方式，尝试在保护模式下调用保护模式函数库来写游戏。

第14章 接口技术

本章导读

对游戏而言接口主要是用来接收用户信息和传输数据的。和主机接口相连的设备包括键盘、鼠标、游戏杆、网卡和调制解调器等。

好的游戏都会选择适合的输入设备。比如，俄罗斯方块游戏比较适合用键盘来操作方块移动和变化；Diabo 则比较多用鼠标来控制主角对象的行为；帝国时代则是键盘加鼠标才能获胜；至于飞行、赛车游戏游戏杆一定使首选设备。

除了输入设备以外，如果我们要进行多台机器的实时游戏则必须通过数据传输接口来实现。比如通过网卡实现局域网中的游戏和利用调制解调器拨号实现电话对打或者 Internet 网络游戏。

本章将主要介绍在游戏中使用适合 C 语言编程的键盘、鼠标读取和串口连接技术。

本章重点

（1）各类键码的读取、键盘缓冲区的清空、多键同时按下检测；
（2）鼠标在程序中的调用、任意鼠标形状的设计、图像文件鼠标的使用；
（3）两机串口的使用（读取、写入），通过串口进行数据传输、串口游戏的实现。

14.1 键　　盘

键盘对于电脑来说无疑是最主要的输入设备之一。在游戏中键盘可以进行较为复杂的用户操作，同时较鼠标而言，键盘对程序在用户输入方面的制作要求要低了很多。我们在游戏中除了要读取键盘的信息以外，还会涉及多键同时按下、模拟键盘输入和清空键盘缓冲等问题。本节就是对这些问题进行讨论。

14.1.1 键盘读取

在谈论键盘之前有必要先介绍两个概念：ASCII 码和扫描码。很多人对 ASCII 码比较熟悉，而不太清楚扫描码的意思。当我们按下键盘的时候，键盘电路中将传送给电脑的不是 ASCII 码而是扫描码。比如，按下 1 键的时候，键盘立即将 1 的扫描码传送给电脑，而电脑再将它转化成对应的 ASCII 码 31h。

1. 扫描码读取

键盘每一个键都有一个扫描码，我们发现扫描码的排列是按照键盘从左到右、从上到下的顺序进行的。拆开键盘看一下，原来键盘的电路是通过纵横交错的线路实现的（如图 14-1）。当有一个按键被按下的时候，此按键所在的横向线路和纵向线路导通，于是在所有原先全部是高电压的横向线路（比如将原先每条线路高电压表示为 0000000000）和纵向线路（比如是 000000）中分别出现了一个低电压（0000000100 和 010000），将两个信号组合

起来就产生了键盘的扫描码。

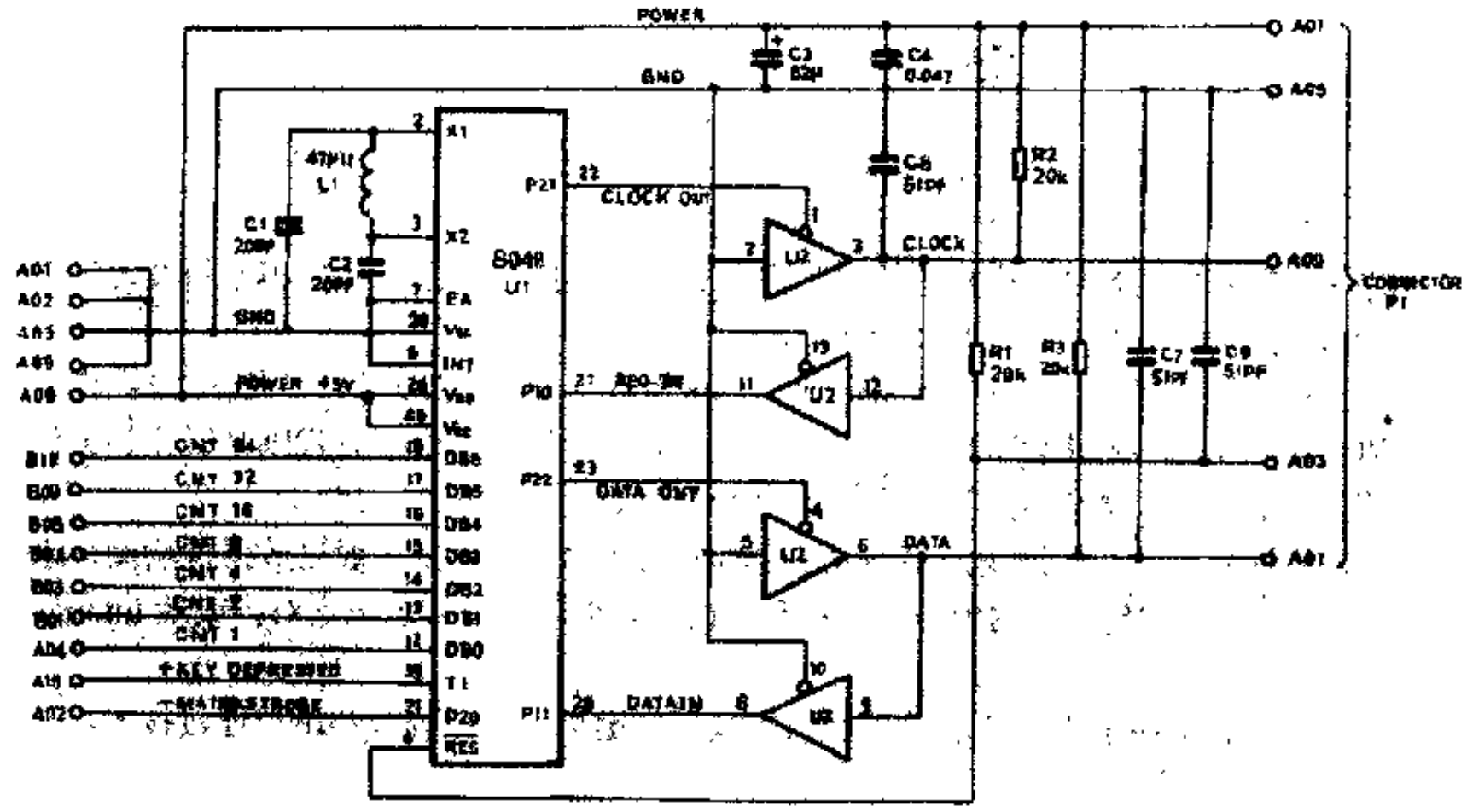


图 14-1　键盘电路示意图

有关键盘扫描码表请查阅附录 D。这里我们介绍使用 BIOS 中断来查询键盘，并且获得其扫描码。键盘 BIOS 中断调用如下：

16H BIOS 中断：

功能：00h——从键盘读取字符串。

功能：01h——读键盘状态。

功能：02h——返回键盘标志。

具体检测键盘的过程如下：

（1）通过触发 BIOS 中断 16H 功能 01H 读取键盘状态；

（2）如果没有按键则返回 0，跳到步骤 5；

（3）否则，通过触发 BIOS 中断 16H 功能 00H 读取扫描码；

（4）清空 AX 寄存器高字节并且返回 AX，结束；

（5）清空 AX 寄存器并且返回 AX；

对于游戏我们希望在不影响其进行的情况下检测是否有键盘按键，以下是一个检测键盘并且获取键盘扫描码的函数：

```
unsigned char Get_Scan_Code(void){
asm mov ah,01h//读键盘状态
asm int 16h//中断 16H
asm jz empty//如果没有按键则跳到 empty
asm mov ah,00h//从前盘读取字符串
asm int 16h//中断 16H
asm mov al,ah//将高字节数据读到低字节
```

```
asm xor ah,ah//清空高字节
asm jmp done//跳到 done
 empty:
asm xor ax,ax//清空 ax 寄存器
 done: }//返回 ax
```

以下是检测某一个具体物理按键是否被按下的函数：

```
int Test_Scan_Code(int scan){ //判断检测到的扫描码是否和给出的扫描码相同
    if(Get_Scan_Code()==scan) return(1);
    else return(0); }
```

具体的扫描码读取例程 scan.c 请查阅所附光盘的“source\14”目录。

为什么我们要在游戏中检测扫描码呢？其实检测 ASCII 码也完全可以解决问题，只是我们发现在玩游戏的时候常常有这样一个情况：游戏让你按某些键进行一些动作或者是退出，可是由于我们不小心按了 Caps Lock 键导致大写而无法对应于相应的操作。我们固然可以采取检测其大小写两个 ASCII 码的方法，但是毕竟这样非常麻烦而且容易出现遗漏。如果我们使用检测扫描码的方法就简单了很多，因为每一个键盘按键无论它处于大写还是小写状态都对应一个扫描码。扫描码只和物理键相关而 ASCII 码是和逻辑键相关的。

2. ASCII 码读取

ASCII 码实际是由键盘扫描码转换而来的。之所以要将扫描码转换成 ASCII 码是因为物理键的数量（100 个左右）是有限的，而我们需要的逻辑键（小于 256 个）的数量却是远远超过物理键实际数量的。在不能扩大物理键的前提下，我们只能采取对物理键（也就是扫描码）进行一些组合来产生更多的逻辑键（也就是 ASCII 码，见附录 D）。比如我们将 Caps Lock 键的两种状态（on 和 down）和许多按键组合就可以产生大小写、数字和各类符号的逻辑键。

对于游戏我们希望在不影响其进行的情况下检测是否有键盘按键，这里给出一个检测并且读取键盘 ASCII 字符的函数：

```
unsigned char Get_Ascii_Key(void){
if(bioskey(1))//检测是否有按键
return(bioskey(0));//如果有按键返回其 ASCII 码
else return(0); }//如果没有按键则返回 0
```

以下是某个 ASCII 码是否被检测到的函数：

```
int Test_Ascii_Key(int ascii){
if(Get_Ascii_Key()==ascii)//判断检测到的 ASCII 码是否和给出的 ASCII 码相同
return(1);
else return(0); }
```

读取 ASCII 码的例程 ascii.c 请查阅所附光盘的“source\14”目录。

3. 控制键读取

键盘有许多控制键无法用读取扫描码和 ASCII 码的方法获得，我们可以使用 bioskey(2) 函数来读取它们。当然我们也可以直接到存放控制键的内存地址去取出两个字节的控制键信息，存放控制键的内存地址是 00417H。见表 14-1。

表 14-1　控制键屏蔽码

Bit	Value	Meaning
0	0x01	RightShiftpressed
1	x02	LeftShiftpressed
2	0x04	Ctrlpressed
3	x08	Altpressed
4	x10	ScrollLockon
5	x20	Num Lockon
6	x40	Capson
7	x80	Inserton
8	x0100	LeftCtrlpressed
9	x0200	LeftAltpressed
10	x0400	RightCtrlpressed
11	x0800	RightAltpressed
12	x1000	ScrollLockpressed
13	x2000	NumLockpressed
14	x4000	CapsLockpressed
15	x8000	SysReqpressed

以下给出控制键读取的函数：

```
unsigned int Get_Control_Keys(unsigned int mask){
delay(15);//延迟一点时间后再读取
return(*shift_key&mask); }//返回被检测的控制键是否被按下的信息
```

其中指针变量 shift_key 定义如下：

```
int far *shift_key=0x00400017;
```

控制键读取的程序 control.c 请查阅所附光盘的“source\14”目录。

14.1.2　同时按下问题

在游戏中我们常常碰到组合键的问题，最记忆犹新的就是 MARIO 里面按住方向键可以前进，但是速度不快，必须按住 Ctrl 才能够加速。同时按下的两个键通常一个是控制键一个是普通键，我们可以使用上面的函数来检测它们。只要当两个键同时被检测到时，成功。

```
int Test_Combination_Keys(unsigned int mask,int key) {
   return(Get_Control_Keys(mask)&&Test_Scan Code(key)); }//要求两个函数都返回 1
```

程序 same.c 请查阅所附光盘的“source\14”目录。

14.1.3　模拟按键

我在调试游戏的时候往往考虑用模拟按键来实现用户输入。要实现键盘的模拟输入，必须了解键盘缓冲区的结构及相应编程机理。键盘缓冲区在内存中的地址分配如表 14-2。

表 14-2　键盘缓冲区在内存中的地址分配

地址		含义
0040：001AH	0040：001BH	存放键盘缓冲区首指针
0040：001CH	0040：001DH	存放键盘缓冲区尾指针
0040：001EH	0040：003EH	键盘缓冲区，存放击键字符

每次击键用 2 字节存放。若为功能键，则第一个字节为 0，第二个字节为该键的扩展码；若为普通键，则第一个字节为该健的 ASCII 码，第二字节为 0。每击一次键，数据送入键盘缓冲区中尾指针指向的单元，同时尾指针加 2；从键盘缓冲区读出键时，数据从缓冲区队列中首指针指示的单元取出，首指针同时加 2。若首尾指针相等，表示缓冲区为空。要实现对键盘缓冲区的自动操作，这里利用 C 语言的对内存直接读库函数 peek()和对内存直接写函数 pokeb()，从而实现模拟键盘输入功能。程序中的关键函数是 keyboard(int choice,char key[])，若 choice 参数不为 0，则 key 字符串中的字符的 ASCII 码逐个送入键盘缓冲区；如果 choice 参数为 0，则将功能键的扩展码送键盘缓冲区主程序运行后，如果 2 秒钟内不通过键盘输入所需的文件名，则调用 keyboard 函数利用模拟键盘输入功能输入 memkey，由此实现程序自动运行。

代码 memkey.c 请查阅所附光盘的“source\14”目录。

14.1.4　清空键盘缓冲

游戏中经常发生由于用户按键过重而造成的键盘缓冲区满（会产生蜂鸣声）或者相应滞后的情况。这是由于我们键盘缓冲区内可以存放 16 个按键，然后依次进行处理，当我们快速按键超过 16 个按键时按键将被键盘缓冲区忽略。有时我们明明只需要按某个键 2 到 3 次就能完成的任务（比如子画面向前走几步），我们多按十几次则当用户坐在电脑前开始睡觉了，游戏主人公还在向前走。这种情况让游戏用户非常恼火，所以我们考虑在每次接受用户键盘信息之后做一下清空键盘缓冲区的工作。

我们已经了解了键盘缓冲区的结构，无非就是两个指针（头、尾指针）和一个缓冲。要清空键盘缓冲只需要将键盘缓冲尾指针指向键盘缓冲首指针，具体做法如下：

（1）到存放键盘缓冲区首指针的地址取得首指针；

（2）将它赋给存放键盘缓冲区首指针的地址。

以下给出清空键盘缓冲区的函数：

```
void Clear_Key_Buffer(void) {
```

```
int offset;
offset=peek(0x40,0x1a);//取得首指针
pokeb(0x40,0x1c,offset); }//键盘缓冲区尾指针指向首指针地址
```

键盘缓冲区的例程 keybuf.c 请查阅所附光盘的“source\14”目录。

这个程序还不能非常明显地显示请空键盘缓冲函数的价值，因为这里没有出现键盘缓冲区满的情况。通常游戏中会在用户输入后进行大量的计算和绘图工作，此时继续按键便无法处理而积压到缓冲区中。所以，我们在使用清空键盘缓冲的函数的同时也尽量提高游戏的效率从而使游戏相应用户更快一些。

14.2　鼠　　标

鼠标是非常好用的游戏输入工具，它比键盘更加贴近用户，所以在大多游戏中都主要使用鼠标作为输入设备。TC 标准函数库没有提供鼠标调用的函数，但是许多程序中都通过 BIOS 中断 33h 编制了鼠标函数。除了检测鼠标一些最常用的标准行为以外，本节还要介绍如何来 diy（do it yourself）一下你自己喜欢样式的鼠标，甚至是使用你喜欢的图片来做鼠标的问题。

14.2.1　鼠标基本函数

鼠标的主要行为包括：

（1）任意方向移动；

（2）单击左键；

（3）单击右键；

鼠标驱动程序在计算机中安装后可以被 BIOS 中断 33h 调用。见表 14-3。

表 14-3　33H 的功能号及其作用

功能号	作用
00h	复位鼠标驱动程序
01h	给出鼠标指针
02h	隐藏鼠标指针
03h	获得指针位置和按钮状态
1Ah	设置灵敏度

下面给出鼠标调用函数：

```
int Squeeze_Mouse(int command,int *x,int *y,int *buttons){
union REGS inregs,outregs;
switch(command){
case MOUSE_RESET: { //复位
inregs.x.ax=0x00;//功能 00h
int86(MOUSE_INT,&inregs,&outregs);//调用鼠标中断 33h
```

```
*buttons=outregs.x.bx;
return(outregs.x.ax);
} break;
case MOUSE_SHOW: { //显示鼠标
inregs.x.ax=0x01;
int86(MOUSE_INT,&inregs,&outregs);
return(1);
} break;

case MOUSE_HIDE: {//隐藏鼠标
inregs.x.ax=0x02;
int86(MOUSE_INT,&inregs,&outregs);
return(1);
}break;
case MOUSE_BUTT_POS: {//按键时查询 x，y 位置
inregs.x.ax=0x03;
int86(MOUSE_INT,&inregs,&outregs);
*x=outregs.x.cx;//返回鼠标 x 位置
*y=outregs.x.dx;//返回鼠标 y 位置
*buttons=outregs.x.bx;//返回鼠标按钮状态
return(1);
} break;
case MOUSE_MOTION_RE.: {//获得按钮状态和 x，y 位置
inregs.x.ax=0x03;
int86(MOUSE_INT,&inregs,&outregs);
*x=outregs.x.cx,
*y=outregs.x.dx;
return(1);
} break;
case MOUSE_SET_SENSITIVITY: {
//设置灵敏度
outregs.x.bx=*x;
outregs.x.cx=*y;
outregs.x.dx=*buttons,
inregs.x.ax=0x1A;
int86(MOUSE_INT,&inregs,&outregs);
return(1);
} break;
```

```
default:break;
} }
```

简单的鼠标测试程序 mouse.c 请查阅所附光盘的“source\14”目录。

14.2.2　改变鼠标形状

1. 鼠标形状和热点的改变

我们在游戏中看到过各种各样的鼠标形状，但通常都是由 16*16 的黑白点组成的。描述鼠标属性的结构体如下：

```
typedef struct{
  int mak[2][16];//鼠标码
  int hor;//热点横坐标
  int ver;//热点纵坐标
}mscurstype;
```

其中第二句表示有 2 套 16*16 点阵（int 型数量为 16 的数组=16*16，2 个这样的数组就是两套）组合构成了鼠标形状。第三和第四句分别是鼠标热点相对于 16*16 点阵的横坐标和纵坐标。关于热点的含义，我们在点击鼠标的时候 16*16 点阵鼠标图形中的一个点被作为鼠标当前在屏幕中的位置返回，这个点就是热点。

以下给出一个箭头形状鼠标的结构赋值：

```
mscurstype arrow={0x3fff,0x1fff,0x0fff,0x07ff,
 0x03ff,0x01ff,0x00ff,0x007f,
 0x003f,0x00ff,0x01ff,0x10ff,
 0x30ff,0xf87f,0xf87f,0xfc3f,
 0x0000,0x4000,0x6000,0x7000,
 0x7800,0x7c00,0x7e00,0x7f00,
 0x7f80,0x7e00,0x7c00,0x4600,
 0x0600,0x0300,0x0300,0x0180,0,0};
```

热点很容易看出，就是最后两个数字(0,0)。值得一提的是这里使用的是 16 来表示 2 套 16*16 点阵的数组。其中，第一套 16*16 点阵数组转化为每行 16 位的二进制如下：

```
0x3fff,0x1fff,0x0fff,0x07ff,                0000001111111111  (0x03ff)
0x03ff,0x01ff,0x00ff,0x007f,                0000000111111111  (0x01ff)
0x003f,0x00ff,0x01ff,0x10ff,                0000000011111111  (0x00ff)
0x30ff,0xf87f,0xf87f,0xfc3f,                0000000001111111  (0x007f)
0011111111111111  (0x3fff)                  0000000000111111  (0x003f)
0001111111111111  (0x1fff)                  0000000011111111  (0x00ff)
0000111111111111  (0x0fff)                  0000000111111111  (0x01ff)
0000011111111111  (0x07ff)                  0001000011111111  (0x10ff)
```

0011000011111111　(0x30ff)　　1111100001111111　(0xf87f)

1111100001111111　(0xf87f)　　1111110000111111　(0xfc3f)

第二套 16*16 点阵数组转化为每行 16 位的二进制如下：

0x0000, 0x4000, 0x6000, 0x7000,　　0x7800, 0x7c00, 0x7e00, 0x7f00,

0x7f80, 0x7e00, 0x7c00, 0x4600,　　0x0600, 0x0300, 0x0300, 0x0180,

0000000000000000　(0x0000)　　0100000000000000　(0x4000)

0110000000000000　(0x6000)　　0111000000000000　(0x7000)

0111100000000000　(0x7800)　　0111110000000000　(0x7c00)

0111111000000000　(0x7e00)　　0111111100000000　(0x7f00)

0111111110000000　(0x7f80)　　0111111000000000　(0x7e00)

0111110000000000　(0x7c00)　　0100011000000000　(0x4600)

0000011000000000　(0x0600)　　0000001100000000　(0x0300)

0000001100000000　(0x0300)　　0000000110000000　(0x0180)

两套点阵都是一个鼠标形状，只是上面用 0 表示，下面用 1 表示，而且上面的鼠标比下面的大。那么上下两套点阵到底是怎样的组合关系呢？以下给处上下两套点阵对应位组合产生屏幕点的组合表 14-4。

表 14-4　两套点阵对应位组合产生的屏幕点

上	0	0	1	1
下	0	1	0	1
屏幕点	黑色	白色	透明	与屏幕背景异或点

于是我们很容易组合出最后的屏幕鼠标形状点，即一个边为黑色内部为白色的箭头图形。这里还给出了一个手指形状鼠标结构和一个“林”字样的鼠标结构。

手形鼠标：

```
mscurstype handcurs={-7681,-7681,-7681,-7681,-7681,-8192,-8192,-8192,
        0,0,0,0,0,0,0,0,
        7680,4608,4608,4608,4608,5119,4681,4681,4681,
        -28671,-28671,-28671,-22767,-32767,-32767,-1,4,0};
```

“林”字：

```
mscurstype lin={0xe3e3,0xe3e3,0xe3e3,0xe3e3,
0x8080,  0x8080,0x8080,0xe3e3,
0xc1c1,  0x8080,0x0000,0x2323,
0xe3e3,0xe3e3,0xe3e3,  0xe3e3,
0x0000,0x0808,0x0808,0x0808,
0x0808,0x3e3e,0x0808,0x0808,
```

0x1c1c, 0x2a2a, 0x4848, 0x0808,

0x0808, 0x0808, 0x0808, 0x0000, 2, 0} ;

请注意，手形鼠标的 2 套点阵我们使用的是 10 进制表示方法，这当然也是可以的。不过还是建议使用 16 进制表示，因为那样转换成二进制比较方便。

下面，我们来尝试做一个任意形状的鼠标。当然也可以是一个字。其过程如下：

（1）将你准备制作的鼠标图形（某个文字）用 16*16 点阵形式用黑白灰点阵图形表示出来；（灰点表示透明）

（2）将黑白灰点阵图形转化为 16*16 的 0（0 对应黑色和白色）和 1（1 对应灰色）数字点阵；

（3）将它转化成 16 个 4 位 16 进制数，这就是第一套点阵数组；

（4）同 2 操作；

（5）将它转化成 16 个 4 位 16 进制数，这就是第二套点阵数组；

（6）在黑白灰点阵图形中找一个你认为比较合适的点作为热点，记录下它在点阵中的坐标；

（7）将上面第 3、4、6 步骤得到的数据对应地赋值给鼠标结构体，并且取一个英文名字。

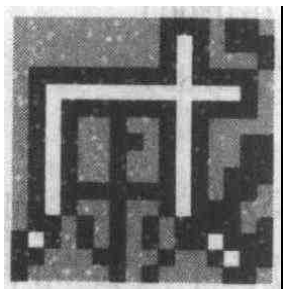


图 14-2　“戚”字点阵图

我有一个好朋友姓“戚”，就尝试用它来做一个鼠标吧！第一步，按照上面的过程我画了一张黑白灰点阵图如图 14-2 所示。

第二步，将它转化成第一套 0、1 数字点阵如下：

1111111110001001
1111111110001110
1000000000000001
1000110110001111
1000110110001100
1000000000001001
1001010101000010
0011100110110000

1111111110001000
1000000000000001
1000000000000001
1000110010001100
1000110110001000
1000110110000011
0001010101000000
0111110101111001

第三步，将这个数字点阵转化成 16 个 4 位 16 进制数如下：

0xff89, 0xff88, 0xff8e, 0x8001,
0x8001, 0x8001, 0x8d8f, 0x8c8c,
0x8d8c, 0x8d88, 0x8009, 0x8983,
0x9542, 0x1540, 0x39b0, 0x7d79,

第四步，将它转化成第二套 0、1 数字点阵如下：

0000000000000000
0000000000100000

0000000000100000
0000000000100000

0011111111111100　　0010000000100000
0010000000100000　　0010000000100000
0010000000100000　　0010000000100000
0010000000100000　　0010000000100000
0000000000000000　　0100000000001000
0000000000000100　　0000000000000000

第五步，将这个数字点阵转化成 16 个 4 位 16 进制数如下：

```
0x0000, 0x0020, 0x0020, 0x0020,
0x3ffc, 0x2020, 0x2020, 0x2020,
0x2020, 0x2020, 0x2020, 0x2020,
0x0000, 0x4008, 0x0004, 0x0000,
```

第六步，我觉得用“戚”字右上角那个点作为热点比较合适，那么热点就是(16,0)。
最后一步，我将这个鼠标形状起名为“qi”赋值如下：

```
mscurstype qi={0xff89, 0xff88, 0xff8, 0x8001,
0x8001, 0x8001, 0x8d8f, 0x8c8c,
0x8d8c, 0x8d88, 0x8009, 0x8983,
0x9542, 0x1540, 0x39b0, 0x7d79,
0x0000, 0x0020, 0x0020, 0x0020,
0x3ffc, 0x2020, 0x2020, 0x2020,
0x2020, 0x2020, 0x2020, 0x2020,
0x0000, 0x4008, 0x0004, 0x0000, 16, 0};
```

2. 鼠标形状设置函数

学会了制作个性化的鼠标形状，我们还需要一个鼠标形状设置函数来将它调入程序中。如果不用这个函数调入，那么即使将你设计的鼠标图形赋值给了结构体，并且起了一个非常好听的名字，程序调用的还是默认的箭头鼠标形状。

以下是鼠标形状设置函数：

```
void setcurshape(mscurstype mask){
  union REGS r;
  struct SREGS s;
  r.x.ax=9;
  r.x.bx=mask.hor;
  r.x.cx=mask.ver;
//设置鼠标的热点
  r.x.dx=FP_OFF(&mask);
  s.es=FP_SEG(&mask);
//设置鼠标的形状
```

```
    int86x(0x33, &r, &r, &s);
}
```

以自行设计的形状显示鼠标：

```
void mscurson(mscurstype shape){
int msvisible;
 union REGS r;
 struct SREGS s;
 setcurshape(shape);//调用设置鼠标形状函数
 r.x.ax=1;
 msvisible=1;
 int86x(0x33,&r,&r,&s); }//显示鼠标
```

第一个函数调用了鼠标 33h 中断中的 9 号功能对数表进行形状和热点设置。具体是在中断前将热点 x 赋值给 bx，热点 y 赋值给 cx，将 2 套点阵数组起始地址和偏移赋值给 dx 和 es。

第二个函数实际就是在调用鼠标 33h 中断 1 号显示鼠标功能函数前调用第一个设置鼠标形状函数，从而完成对鼠标的形状设置和显示。

在程序中我们只需要如下一句调用便能完成形状设置了：

```
mscurson(arrow); //arrow 为赋值箭头形状的结构
```

百变鼠标的例程 mshape.c 请查阅所附光盘的“source\14”目录。其效果如图 14-3、14-4 和图 14-5 所示。

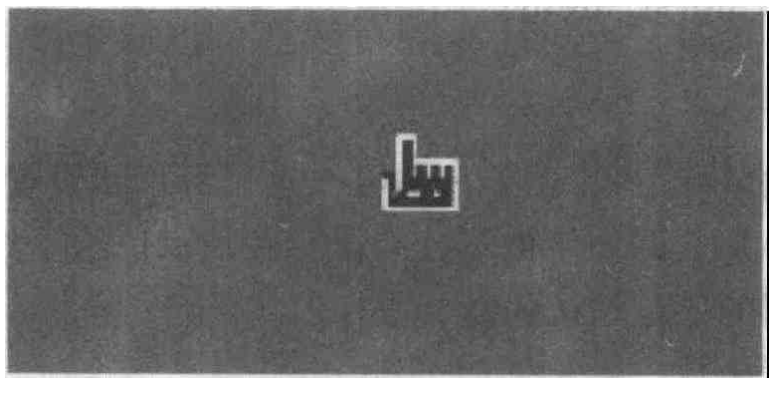


图 14-3　手形鼠标

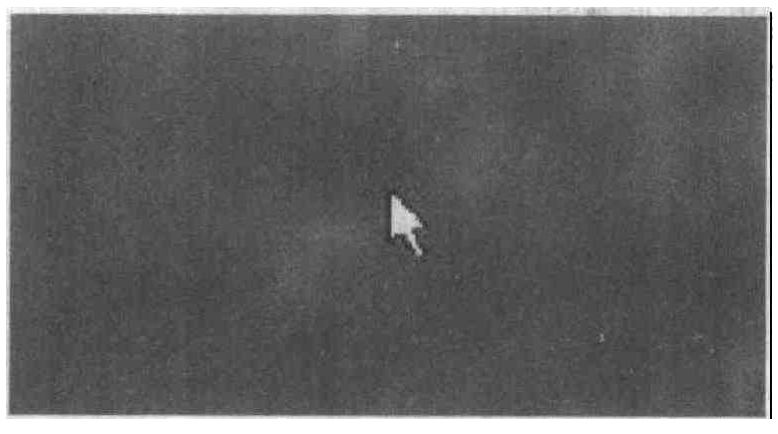


图 14-4　箭头鼠标

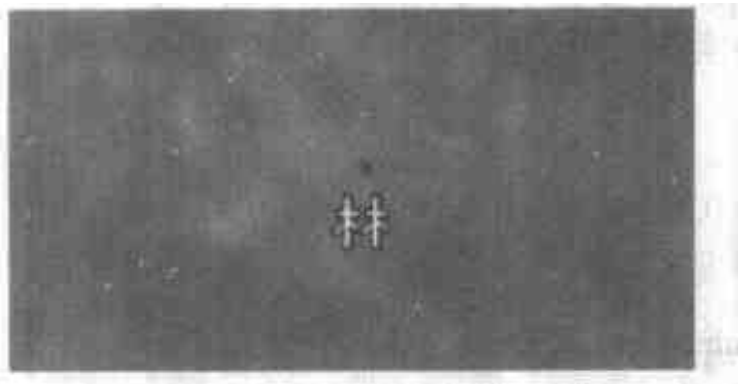


图 14-5　“林”字鼠标

14.2.3　用 pcx 图像做鼠标

虽然我们能够通过改变鼠标形状来实现更多样式的鼠标，但是由于颜色和 16*16 点阵表现空间的限制还不足以满足我们的想象力。这里我使用 pcx 图片绘制了一个如图 14-6 的 65*55 的 256 色玫瑰花鼠标。

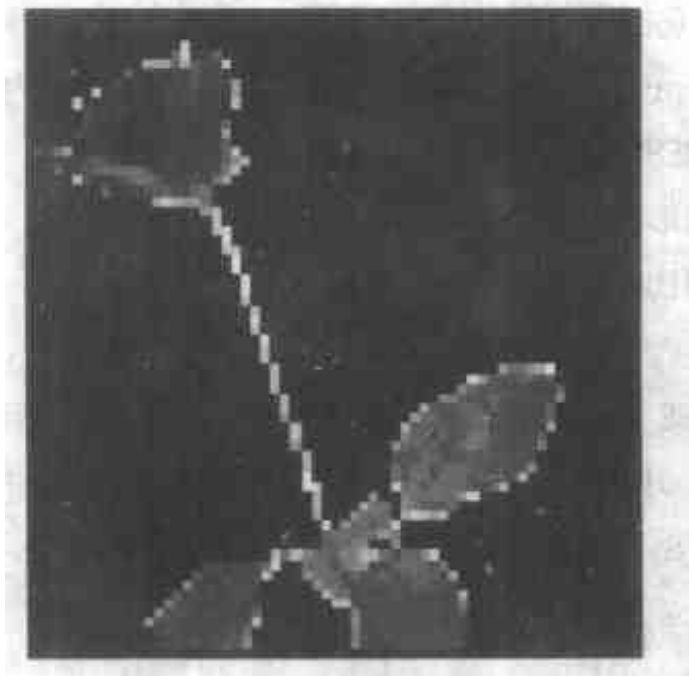


图 14-6　256 色玫瑰鼠标

使用 pcx 图片来做鼠标非常简单，其过程如下：

（1）调用用来制作鼠标的 pcx 图片到屏幕；

（2）将屏幕图像保存到一个子画面（pcx 鼠标图形）内存中；

（3）创建屏幕场景；

（4）保存子画面（pcx 鼠标图形）后屏幕内容；

（5）初始化鼠标；

（6）隐藏鼠标；

（7）进入主游戏循环；

（8）取鼠标位置；

（9）判断用户是否移动过鼠标，如果没有移动过，返回 8；

（10）显示子画面（pcx 鼠标图形）后面的背景内容；

（11）保存子画面（pcx 鼠标图形）新位置（鼠标当前位置）后面的背景内容；

（12）显示子画面（pcx 鼠标图形）于鼠标当前位置；

（13）返回 8，直到退出程序标志出现。

以下给出具体的程序实现过程段：

```
void main(void) {
int xmouse=0, ymouse=0, btnmouse=0, xmouse_old=0, ymouse_old=0;
sprite small;
pcx_picture background_pcx, objects_pcx;
Set_Video_Mode(VGA256);
PCX_Load1("flowersc.pcx", (pcx_picture_ptr)&objects_pcx, 1);//调用图片文件
Sprite_Init_Size((sprite_ptr)&small, 0, 0, 0, 0, 0, 0, 65, 55, 1);//初始化小图片
PCX_Grab_Bitmap_Size_Screen((pcx_picture_ptr)&objects_pcx, (sprite_ptr)&small, 0, 0, 0, 65
,55);  //将屏幕内容复制给小图片
Fill_Screen(0);
PCX_Load1("poem.pcx", (pcx_picture_ptr)&objects_pcx, 1);//背景
Behind_Sprite_Size((sprite_ptr)&small, 65, 55);//取图片后内容
Squeeze_Mouse(MOUSE_RESET, 0, 0, 0);//初始化鼠标
mscurson();//设置鼠标形状
Squeeze_Mouse(MOUSE_HIDE, 0, 0, 0);//隐藏鼠标
while(!kbhit()) {
    Squeeze_Mouse(MOUSE_BUTT_POS, &xmouse, &ymouse, &btnmouse);//取鼠标位置
    if(xmouse!=xmouse_old||ymouse!=ymouse_old) {//判断是否移动过鼠标
        Erase_Sprite_Size((sprite_ptr)&small, 65, 55);//显示小图片后背景内容
        small.x=xmouse/2;
        small.y=ymouse;
        Behind_Sprite_Size((sprite_ptr)&small, 65, 55);//保存小图片后背景内容
        Draw_Sprite_Size((sprite_ptr)&small, 65, 55);//显示小图片于鼠标位置
        xmouse_old=xmouse;
```

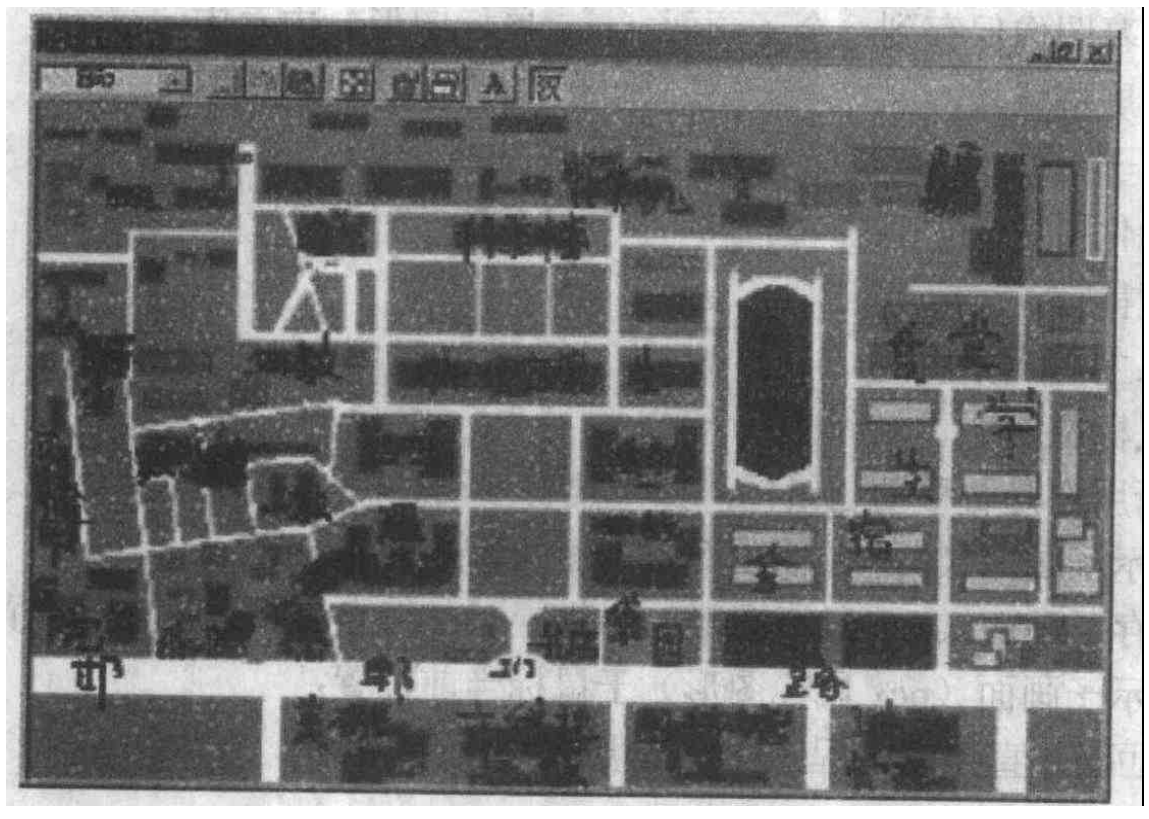


图 14-7　在背景中加入 pcx 图片鼠标后

```
        ymouse_old=ymouse;
    } } getch();
Sprite_Delete_Size((sprite_ptr)&small,1);
Set_Video_Mode(TEXT_MODE); }
```

完整的 pcx 鼠标显示程序 pcxmouse.c 请查阅所附光盘的“source\14”目录。其效果如图 14-7 所示。

14.3　串　　口

14.3.1　串口基础

几乎所有的主机都有两个串口，一个是 9 针的，还有一个是 25 针的（新出来的计算机可能没有 25 针的）。不过要买到一条串行线已经是很难的事情了（我跑了好多配件商店，所有的人都拿出并线给我）。串口程序中实际上只使用到串口的 3 根线，一根是数据发送线，一根是数据接受线还有一根是接地线。表 14-5 是 9 针和 25 针串口 3 根线的序号。

表 14-5　9 针与 25 针串口线作用

串口类型	针号	作用
9 针	2	数据发送线
	3	数据接收线
	5	接地线
15 针	2	数据接收线
	3	数据发送线
	7	接地线

如果我们在将两台电脑连接时真的买不到串口线只需要随便拿三根电线将一个电脑串口的数据发送线连到另一个的数据接收线，然后接地线相连或不连就成了。

此外，许多写游戏的同学由于串口游戏的调试受到两台电脑的硬件要求而不得不放弃。那么只有一台电脑是否能对串口联机程序进行调试了？答案是肯定的。我曾经截下了一个串口鼠标的除鼠标外的串口接头和线，发现里面有 4 根线，当然对我们来说只有 3 根是有用的。只需要将鼠标串口接头和线的另一端做一下导通试验就可以知道哪几个颜色的线是 2、3、5 号线了。然后就是将 2、3 号线连接、5 号线不理睬，最后将串口头插到 9 针的串口就搞定了。知道么，现在你的电脑就成了两台电脑了，因为你从串口送出的数据又从串口的数据接收线回来了。现在可以开始我们的串口程序了。

通过串口的传输字符数据的过程如下：

（1）初始化 A 机和 B 机的串口；

（2）改写串口中断服务函数；

（3）A 机发送数据到串口；

（4）B 机中断函数不断检测串口，如果有数据就放入缓冲数组；

（5）B 机检测缓冲数组中是否有数据；

（6）A 机在传送完所有数据后发送一个结束符号给 B 机；

（7）A 机和 B 机关闭串口，同时恢复原先的中断函数入口。

1. 初始化串口

初始化串口最主要的任务有以下几点：

（1）设置传输基本参数：包括传输时的波特率，发送时的奇偶、停止位数和数据位数；

（2）设置允许中断：允许异步接收发送器（UART）发出中断和串口接收字符时发生中断；

（3）改写串口中断函数：在串口接收到字符后产生的中断中将数据放入缓冲数组；

（4）设置 PIC 中断屏蔽寄存器：将对应的串口中断打开。

任务 1、2 主要是针对 UART 的 9 个寄存器进行的。如果要改变传输的波特率，我们首先需要将 UART 的 3 号寄存器的功能位 7 设定为 1。然后再将波特率的除数（针对 115200 基本波特率而言的）放入 UART 的 8 号寄存器。如果要设定发送奇偶、停止位数和数据位数则需要对 UART 的 3 号寄存器的对应功能位进行设定。

任务 3 中只需要在保存串口中断入口后将改写的中断函数放入，串口中断端口如表 14-6。

表 14-6　串口号及对应中断

串口号	中断端口
COM1&COM3	0x0C
COM2&COM4	0x0B

任务 4 是针对 PIC 的中断屏蔽寄存器(IMR)进行的（详细可以参考有关《微型机算计接口技术》的相关内容）。IMR 寄存器端口是 0x21，寄存器 8 位中位 3 对应 COM2&COM4，位 4 对应 COM1&COM3，并且对应位为 0 时表示中断打开。于是如果要将 COM1 的中断打开只需要将读取的 IMR 寄存器内容和 EF 进行位与就可以了。

以下是初始化串口的函数：

```
Open_Serial(int port_base, int baud, int configuration){
open_port = port_base;//设置当前使用的基串口号，COM1 or COM2
outp(port_base + SER_LCR, SER_DIV_LATCH_ON);//寄存器 3 位 7，允许改变波特率
outp(port_base + SER_DLL, baud);//寄存器 8，波特率除数锁存低字节为 baud，如果 baud 为 12，则
//波特率为 115200/12=9600
outp(port_base + SER_DLH, 0);//寄存器 8，波特率除数锁存高字节为 0
outp(port_base + SER_LCR, configuration);
//寄存器 3，线路控制寄存器，控制发送奇偶、停止位数和数据位数
outp(port_base + SER_MCR, SER_GPO2);//寄存器 4，允许 UART
//发出中断，也就是为接收是产生中断铺垫
outp(port_base + SER_IER, 1);//寄存器 1，每次从串口接收一个字符发生一次中断
```

```
if (port_base == COM_1) {//判断串口号
   Old_Isr = _dos_getvect(INT_SER_PORT_0);//读取 COM1 或者 COM2 的中断服务函数入口
   _dos_setvect(INT_SER_PORT_0, Serial_Isr);//设置串口接收数据到缓冲中断
   printf("\nOpening Communications Channel Com Port #1...\n");
   } else {
   Old_Isr = _dos_getvect(INT_SER_PORT_1);
   _dos_setvect(INT_SER_PORT_1, Serial_Isr);
   printf("\nOpening Communications Channel Com Port #2...\n"); }
old_int_mask = inp(PIC_IMR);//读取旧的中断屏蔽寄存器内容
outp(PIC_IMR, (port_base==COM_1) ? (old_int_mask & 0xEF) : (old_int_mask & 0xF7 )); }
//将 PIC 的 IMR 寄存器设置为对应串口中断开
```

2. 串口中断

由于在初始化串口时设定了串口在每次接收到一个字符的时候产生一次中断，于是改写串口的中断服务函数的目的主要就是将这个得到的字符放入缓冲数组。如果不改写串口中断函数而让串口缓冲读取函数直接读取串口送来的字符，则将流失许多从串口送来的字符，解决的办法就是通过中断函数将所有发来的字符放入缓冲，等待串口缓冲读取函数到缓冲中去。

这个缓冲数组实际上就是一个先进先出的堆栈。如同键盘缓冲一样，我们只需要在开辟一定空间的基础上设定两个指针，一个指向最先放入缓冲的数据地址，一个指向最后放入缓冲中的数据地址，它们分别被称为头指针和尾指针。每次发生串口中断时由于缓冲数据增加了 1，于是尾指针就要向后移动 1 位。同样在我们读取缓冲数据的时候由于缓冲数据将减少 1，就将头指针向后移动 1 位。

如果出现串口中断写缓冲和串口读取缓冲同时，即我们可能同时要对缓冲进行读和写的操作，这是非常危险的，所以我们建议设置一个锁存缓冲区的标志来解决问题。

以下是改写的串口中断服务程序的函数：

```
void interrupt far Serial_Isr() {//串口接收数据到缓冲中断服务函数
serial_lock = 1;//锁缓冲，不允许读串口函数读取数据
ser_ch = inp(open_port + SER_RBF);//寄存器0，接收字符
if (++ser_end > SERIAL_BUFF_SIZE-1)//设置缓冲数据尾指针到下一个数据
ser_end = 0;
ser_buffer[ser_end] = ser_ch;//将传来的字符放入缓冲数组
++char_ready;//缓冲数组中字符数量加 1
outp(PIC_ICR,0x20);//可以省略，允许 PIC 发出一次中断
serial_lock = 0; }//打开缓冲
```

3. 关闭串口

关闭串口实际上就是将初始化串口时的设定恢复到原来状态，最主要的任务有以下几点：

（1）设置不允许中断：不允许 UART 发出中断，同时不允许串口接收字符时发生中断；

（2）恢复 PIC 的中断屏蔽寄存器：将对应的串口中断关闭。

（3）恢复串口中断函数：恢复原先的串口中断服务函数入口；

以下是关闭串口的函数：

```
Close_Serial(int port_base){
outp(port_base + SER_MCR, 0);//寄存器 4 恢复，不允许 UART 送出中断
outp(port_base + SER_IER, 0);//寄存器 1 恢复，不允许串口在接受字符时产生中断
outp(PIC_IMR, old_int_mask );//恢复 PIC 的 IMR 寄存器原先的数据
if (port_base == COM_1) {
   _dos_setvect(INT_SER PORT_0, Old_Isr);//恢复 COM1 或者 COM2 中断函数入口
   printf("\nClosing Communications Channel Com Port #1.\n");
   } else {
   _dos_setvect(INT_SER_PORT_1, Old_Isr);
   printf("\nClosing Communications Channel Com Port #2.\n");
   } }
```

4. 写串口

写串口的工作非常简单，有以下几步：

（1）等待前一个输出数据已被处理；

（2）不允许中断发生；

（3）将字符放入 UART 的传输保存寄存器。

（4）允许中断发生。

步骤 1 是为了防止传输保存寄存器中内容还没有被处理，从而导致输出到串口的数据被覆盖而产生遗漏。通过检测 UART 寄存器 5 的位 5 来确定其已经放入输出移位寄存器，如果没有放入就不断循环判断直到其放入。

关闭中断的原因是我们担心在向串口写字符的时候突然从另一台机器上发送来一个字符到串口。此时 UART 的写缓冲寄存器和读缓冲寄存器是共用 0 号寄存器的，关闭中断后就不可能从串口读到字符了，于是就可以安心地将字符放入 UART 写缓冲寄存器。

以下是写串口的函数：

```
Serial_Write(char ch){
while(!(inp(open_port + SER_LSR) & 0x20)){}
//寄存器 5 位 5，判断前面传输缓冲内容是否被放入输出移位寄存器
asm cli//中断标志设置为 0，不允许中断
outp(open_port + SER_THR, ch);//寄存器 0，将字符放入传输保存寄存器
asm sti }//开中断
```

5. 读串口缓冲

读串口缓冲正好与串口中断函数写串口缓冲的过程对应。具体过程如下：

（1）在读取缓冲之前都要循环判断目前是否处于串口中断函数锁存缓冲的状态；

（2）不处于锁存的状态时开始判断缓冲内是否还有数据；

（3）如果有数据，从缓冲头指针位置读取一个字符；

（4）缓冲的头指针都向后移动 1 位，缓冲数据量标志减 1。

以下是读串口的函数：

```
int Serial_Read(){
int ch;
while(serial_lock){}//中断函数是否处于锁缓冲区状态，中断函数不再进行
if (ser_end != ser_start) {//检测缓冲数组中是否有内容
   if (++ser_start > SERIAL_BUFF_SIZE-1)//设置缓冲数据头指针到下一个数据
   ser_start = 0;
   ch = ser_buffer[ser_start];//读取当前缓冲内容
   if (char_ready > 0)//判断缓冲是否有内容
   --char_ready;//缓冲内数据量减 1
   return(ch);//返回读取的字符
   } else return(0); }//没有字符在缓冲则输出 0
```

以下判断串口读缓冲区是否准备好了的函数：

```
int Ready_Serial() {
return(char_ready); }//返回缓冲内数据量，大于 0 则表示准备好了
```

以下给出串口字符传送基本程序段：

```
...//变量设定
Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
//打开 COM1，设定波特率为 9600 字节、没有奇偶、8 位数据、1 位停止位
while(!done) {//进入传输循环
 if (kbhit()) {//判断是否有按键
ch = getch();//取得输入的字符
printf("%c",ch);
Serial_Write(ch);//将字符写入串口
if (ch==27) done=1;//如果字符是 ESC 就退出程序
if (ch==13) {//判断字符是否是回车
printf("\n");
Serial_Write(10); } }
 if (ch = Serial_Read())//判断串口缓冲是否有字符来
printf("%c", ch);
 if (ch == 27) {//判断从串口接收到的数据是否是 ESC
```

```
printf("\nRemote Machine Closing Connection.");
done=1;//如果时就退出
} } Close_Serial(COM_1);//关闭 COM1
...
```

14.3.2　利用串口传输文件

在没有网卡只有一根并行线的时候，常常用 Windows 下的两机连接或者 DOS 下的 interlnk.exe 和 intersvr.exe 在两台机器间传输文件。由于前一节我们学会了串口传输最基本的函数，现在希望能够尝试用串行线来传输任意文件。具体的思路如下：

（1）初始化 A 机和 B 机串口；

（2）在 A 机打开一个要传输的文件，B 机打开一个接收传输的文件；

（3）从 A 机向 B 机发送一个开始传送的标志符号；

（4）B 机在读取到开始发送的信息后进入主循环；

（5）A 机从文件每次读取一个字符到串口；

（6）B 机不断读取字符到文件；

（7）A 机文件读取结束后完成传输；

（8）A 机和 B 机关闭串口。

串口传输文本文件的程序包括发送文件程序 send1.c 和接收文件程序 get1.c，分别请查阅所附光盘的“source\14”目录。这里只需要修改主函数即可，主要代码如下：

```
void Read_Serial_File(FILE *name) {
char ch=' ',press;
int done=0;
FILE *fp;
   if((fp=fopen(name,"w+b"))==NULL) {//新建或者打开文件，写方式
        printf("cannot open the file\n");
        exit(0);
   } fseek(fp, 0, SEEK_SET);//偏移到文件头
//打开串口
Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
while(ch!=Serial_Read());//接受初始符号
while(!done) {//接受串口数据直到对方电脑要求停止
   if(ch=Serial_Read()) {//从串口读取字符到 ch
        if (ch==27) {//如果读取到结束符号
             done=1;
             break;
        } printf("%c",ch);//将 ch 显示到屏幕
        fputc(ch,fp);//将 ch 写入文件
   } } Close_Serial(COM_1);//关闭串口
```

```
fclose(fp); }
void main() {
char *getfile;
getfile="test.c";//文件名称
Read_Serial_File(getfile); }//读串口来的内容到文件
```

在两台用串口连接的机器上分别运行以上两个程序可以实现任意文本文件的传输。如果要进行 2 进制文件（例如 EXE 文件）的传输，我们就不能用这种方式了，因为 2 进制文件不是如同文本文件那样的只使用标准 ASCII 码，而是包括了扩展 ASCII 码，于是特殊符号在 2 进制文件中都出现导致文件在传输中意外结束。这里采用一个愚蠢但有趣的（但是低效的方法）来实现传输 2 进制文件，我们读出 2 进制文件的每个字节的每个位，然后传输它的位（也就是 0 和 1），此后在另一台机器上重新将位拼成字节。

向串口写入 2 进制文件的程序部分代码 send2.c：

```
void Send_Serial_BFile(FILE *name) {
FILE *file;
char *buf;
unsigned char mask[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
char ch,press;
int done=0,j;
FILE *fp;
   if((fp=fopen(name,"r+b"))==NULL) {//打开 2 进制文件
        printf("cannot open the file\n");
        exit(0);
   } fseek(fp, 0, SEEK_SET);//偏移到文件头
//打开串口
Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
printf("press any key to begin sending");
getch();
Serial_Write(' ');//发送初始信号
while(!done&&!feof(fp)) {
   fread(buf,1,1,file);//读一个 2 进制字符
   if(feof(file)) Serial_Write(27);//判断文件是否结束
   for(j=0;j<8;j++) {//传输此 2 进制字符的每一个位
        if(buf[0]&mask[j]) {//从高位开始检测
             Serial_Write('1');//如果该位是 1，则传输 1 到串口
             printf("1");
        } else Serial_Write('0');// 如果该位是 0，则传输 0 到串口
             printf("0"); }
   if (kbhit()) {//判断是否有按键
```

```
        press=getch();
        if (press==27) {
            Serial_Write(27);
            done=1;
        } } } Close_Serial(COM_1);//关闭串口
fclose(fp); }
void main(int argc,char *argv[]){
argc=2;
argv[1]="mouse.exe";
   if(argc<2)    {
        printf("\nUsage:display filename.wav!!!");
        exit(0);
   } Send_Serial_BFile(argv[1]); }//向串口传输 2 进制文件
```

从串口读取 2 进制文件的程序部分代码 get2.c：

```
void Read_Serial_BFile(FILE *name1,FILE *name2) {
char ch=' ',press;
int done=0;
FILE *fp;
FILE *file;
FILE *stream;
int i,j;
char *buf;
char *buf2;
unsigned char mask[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
   if((fp=fopen(name1,"w+b"))==NULL) {//打开临时文件，写方式
        printf("cannot open the file\n");
        exit(0);
   } fseek(fp, 0, SEEK_SET);//偏移到文件头
//打开串口
Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
while(ch!=Serial_Read());//读取初始信号
while(!done) {
   if(ch=Serial_Read()) {//从串口读取字符到 ch
        if (ch==27) {//判断是否是结束符号
            done=1;
            break;
       } printf("%c",ch);
        fputc(ch,fp);//将读取的数据（0 和 1）写入临时文件
```

```
    } } Close_Serial(COM_1);//关闭串口
fclose(fp);//关闭临时文件
    file=fopen(name1,"rb");//打开临时文件，读方式
    stream=fopen(name2,"w+b");//打开最终 2 进制文件，写方式
    for(i=0;!feof(file);i++) {//将临时文件中 0 和 1 通过重新组合恢复到 2 进制文件
        buf2[0]=0x00;
        for(j=0;j<8;j++) {//临时文件 8 个字符恢复成 1 个字符
            fread(buf,1,1,file);//从临时文件读取一个字符
            if(buf[0]=='1') {
                printf("1");
                buf2[0]=buf2[0]|mask[j];//如果是 1 则该位为 1
            } else printf("0");
        } printf(" ");
        if(!feof(file))
            fwrite(buf2, 1, 1, stream);//将处理后的字符放入最终文件
    } fclose(file);//关闭临时文件
    fclose(stream); }//关闭 2 进制文件
void main() {
char *getfile,*getfile2;
getfile="test.c";//临时文件
getfile2="mouse.exe";//最终 2 进制文件
Read_Serial_BFile(getfile,getfile2); }//从串口读 2 进制文件
```

14.3.3　两机坦克对打例程

在第 10 章子画面技术中给出过一个简单的坦克对打游戏例程。当时两个坦克的控制者都是在同一台机器上进行的。现在我们通过串口调用来实现在两台电脑上进行坦克对战。

要将游戏改成两台电脑联机游戏需要改进以下几点：

（1）驱动串口；

（2）将一辆坦克（另一台机器控制）输入设备的来源从键盘改变成串口；

（3）将另一辆坦克（本地机器控制）从键盘获得的控制信息复制到串口；

（4）游戏结束后关闭串口。

原先在游戏中读取两辆坦克键盘控制信息的时候我们采用的是：

```
if(kbhit())
...
switch(get_key=getch())
...
```

现在只需要将其中一个改成：

```
if(get_line=Serial_Read())
```

```
...
switch(get_line)
...
```

此外，另一个仍然没有改变键盘控制的坦克在接收到键盘信息的同时，用写串口函数向串口输出获得的控制信息：

```
Serial_Write(get_key);
```

这样就基本完成了两机坦克对打游戏的功能了。不过在实际操作中我们有时还是会碰到一个同步的问题，由于两台机器的速度性能差距较大，导致两台电脑处理游戏的速度产生了先后。结果 A 机的 1 号坦克已经走到中间，而 B 机上显示它还刚刚从左端开始向中间移动。虽然，这种情况下我们接收的字符没有遗失，但滞后的情况的确需要解决。办法就是再增加一个远端机器开始处理画面动作的回馈，只有接收到回馈后才能进行相应的画面动作。这样，两台机器可以在非常接近的时间内开始动作，而以后的所有动作都需要等双方共同开始。

以下是主程序段(本串口坦克子画面对象处理没有使用第 10 章子画面计数中的对象思路)：

```
void main(void){
long index;
sprite tank1,tank2;
bullet bullet1,bullet2;
pcx_picture background_pcx,objects_pcx;
int tank1_direction=0,tank2_direction=0,done=0;
float dx,dy,dxx=0,dyy=0,angle;
float dx1,dy1,dxx1=0,dyy1=0,angle1;
int song_flag=0;
char get_key,get_line,old_line;
int i,j=2,k,k1,l=1,m,n;
bullet1.flag=0;
bullet2.flag=0;
bullet1.x=bullet1.y=400;
bullet2.x=bullet2.y=400;
Set_Video_Mode(VGA256);
//初始化 com1，打开串口
Open_Serial(COM_1,SER_BAUD_9600,SER_PARITY_NONE | SER_BITS_8 | SER_STOP_1);
Blit_String(90,40,7,"A T T A N K !!!",1);
PCX_Init((pcx_picture_ptr)&objects_pcx);
getch();
for(index=0;index<300000;index++,Plot_Pixel_Fast(rand()%320,rand()%200,0));
```

```
PCX_Load_Screen("poem.pcx", (pcx_picture_ptr)&objects_pcx, 1);
PCX_Load("attank2.pcx", (pcx_picture_ptr)&objects_pcx, 1);
 Sprite_Init((sprite_ptr)&tank1, 0, 0, 0, 0, 0, 0);
for(index=0;index<MAX_SPRITE_FRAMES;index++){
 PCX_Grab_Bitmap((pcx_picture_ptr)&objects_pcx, (sprite_ptr)&tank1, index, index, 0); }
Sprite_Init((sprite_ptr)&tank2, 0, 100, 0, 0, 0, 0);
for(index=0;index<MAX_SPRITE_FRAMES;index++){
PCX_Grab_Bitmap((pcx_picture_ptr)&objects_pcx, (sprite_ptr)&tank2, index, index, 1); }
PCX_Delete((pcx_picture_ptr)&objects_pcx);
tank1.curr_frame=tank1_direction;
tank2.curr_frame=tank2_direction;
Behind_Sprite((sprite_ptr)&tank1);
Behind_Sprite((sprite_ptr)&tank2);
while(!done){
l=-1;
k=0;
k1=0;
Erase_Sprite((sprite_ptr)&tank1);
Erase_Sprite((sprite_ptr)&tank2);
if(bullet1.flag==1){
Plot_Pixel_Fast(bullet1.x, bullet1.y, bullet1.color_gd);
bullet1.x+=bullet1.direction.x*16;
bullet1.y+=bullet1.direction.y*16;
if(bullet1.x>336 ||bullet1.x<-16||bullet1.y>184||bullet1.y<-16){
bullet1.flag=0;
bullet1.x=bullet1.y=400; } }
if(bullet2.flag==1){
Plot_Pixel_Fast(bullet2.x, bullet2.y, bullet2.color_gd);
bullet2.x+=bullet2.direction.x*16;
bullet2.y+=bullet2.direction.y*16;
if(bullet2.x>=320||bullet2.x<=0||bullet2.y>=200||bullet2.y<=0) {
bullet2.flag=0;
bullet2.x=bullet2.y=400; } }
if(l==1){
if(kbhit()) {//读取本地按键
k=1;
dx=dy=0;
k1=1;
```

```
dx1=dy1=0;
switch(get_key=getch()){
case '8':
tank1_direction=0;
angle=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank1_direction);
dxx=dx=TANK_SPEED*cos(PI*angle/180);
dyy=dy=TANK_SPEED*sin(PI*angle/180);
break;
case '6':
tank1_direction=1;
angle=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank1_direction);
dxx=dx=TANK_SPEED*cos(PI*angle/180);
dyy=dy=TANK_SPEED*sin(PI*angle/180);
break;
case '2':
tank1_direction=2;
angle=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank1_direction);
dxx=dx=TANK_SPEED*cos(PI*angle/180);
dyy=dy=TANK_SPEED*sin(PI*angle/180);
break;
case '4':
tank1_direction=3;
angle=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank1_direction);
dxx=dx=TANK_SPEED*cos(PI*angle/180);
dyy=dy=TANK_SPEED*sin(PI*angle/180);
break;
case '0':
if(bullet1.flag==0){
song_flag=1;
bullet1.flag=1;
if(tank1_direction==0){
bullet1.direction.y=-1;
bullet1.direction.x=0;
bullet1.x=tank1.x+7;
bullet1.y=tank1.y-1;
} else if(tank1_direction==1) {
bullet1.direction.x=1;
bullet1.direction.y=0;
```

```
bullet1.x=tank1.x+16;
bullet1.y=tank1.y+7; }
else if(tank1 direction==2) {
bullet1.direction.y=1;
bullet1.direction.x=0;
bullet1.x=tank1.x+8;
bullet1.y=tank1.y+16;
} else if(tank1_direction==3) {
bullet1.direction.x=-1;
bullet1.direction.y=0;
bullet1.x=tank1.x-1;
bullet1.y=tank1.y+8;
} bullet1.color=15;
}break;
case 'q':
done=1;
break;
default:break; }
tank1.x+=(int)(dx+.5);
if(tank1.x>319-(int)SPRITE_WIDTH)
tank1.x=319-(int)SPRITE_WIDTH;
else if(tank1.x<0)
tank1.x=0;
tank1.y-=(int)(dy+.5);
if(tank1.y>199-(int)SPRITE_HEIGHT)
tank1.y=199-(int)SPRITE_HEIGHT;
else if(tank1.y<0)
tank1.y=0;
tank1.curr_frame=tank1_direction;
Serial_Write(get_key); }//将按键发送到串口
if(k==0){
tank1.x+=(int)(dxx+.5);
if(tank1.x>319-(int)SPRITE_WIDTH)
tank1.x=319-(int)SPRITE_WIDTH;
else if(tank1.x<0)
tank1.x=0;
tank1.y-=(int)(dyy+.5);
if(tank1.y>199-(int)SPRITE_HEIGHT)
```

```
tank1.y=199-(int)SPRITE_HEIGHT;
else if(tank1.y<0)
tank1.y=0;
tank1.curr_frame=tank1_direction; }
if(get_line=Serial_Read()) {//读取串口数据，远端机器按键
//处理远端机器控制的坦克画面在本地机器的显示
k1=1;
switch(get_line){
case '8':
tank2_direction=0;
angle1=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank2_direction);
dxx1=dx1=TANK_SPEED*cos(PI*angle1/180);
dyy1=dy1=TANK_SPEED*sin(PI*angle1/180);
break;
case '6':
tank2_direction=1;
angle1=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank2_direction);
dxx1=dx1=TANK_SPEED*cos(PI*angle1/180);
dyy1=dy1=TANK_SPEED*sin(PI*angle1/180);
break;
case '2':
tank2_direction=2;
angle1=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank2_direction);
dxx1=dx1=TANK_SPEED*cos(PI*angle1/180);
dyy1=dy1=TANK_SPEED*sin(PI*angle1/180);
break;
case '4':
tank2_direction=3;
angle1=(90+360-360/(MAX_SPRITE_FRAMES/2)*(float)tank2_direction);
dxx1=dx1=TANK_SPEED*cos(PI*angle1/180);
dyy1=dy1=TANK_SPEED*sin(PI*angle1/180);
break;
case '0':
if(bullet2.flag==0){
song_flag=1;
bullet2.flag=1;
if(tank2_direction==0){
bullet2.direction.y=-1;
```

```
bullet2.direction.x=0;
bullet2.x=tank2.x+7;
bullet2.y=tank2.y-1; }
else if(tank2_direction==1){
bullet2.direction.x=1;
bullet2.direction.y=0;
bullet2.x=tank2.x+16;
bullet2.y=tank2.y+7; }
else if(tank2_direction==2) {
bullet2.direction.y=-1;
bullet2.direction.x=0;
bullet2.x=tank2.x+8;
bullet2.y=tank2.y+16;
} else if(tank2_direction==3) {
bullet2.direction.x=-1;
bullet2.direction.y=0;
bullet2.x=tank2.x-1;
bullet2.y=tank2.y+8; }
bullet2.color=7;
}break;
case 'q':
done=1;
break;
default:break; }
tank2.curr_frame=tank2_direction;
tank2.x+=(int)(dx1+.5);
tank2.y-=(int)(dy1+.5);
if(tank2.x>(319-(int)SPRITE_WIDTH))
tank2.x=319-(int)SPRITE_WIDTH;
else if(tank2.x<0)
tank2.x=0;
if(tank2.y>(199-(int)SPRITE_HEIGHT))
tank2.y=199-(int)SPRITE_HEIGHT;
else if(tank2.y<0)
tank2.y=0;
} else if(k1==0) {
tank2.x+=(int)(dxx1+.5);
if(tank2.x>319-(int)SPRITE_WIDTH)
```

```
tank2.x=319-(int)SPRITE_WIDTH;
else if(tank2.x<0)
tank2.x=0;
tank2.y-=(int)(dyy1+.5);
if(tank2.y>199-(int)SPRITE_HEIGHT)
tank2.y=199-(int)SPRITE_HEIGHT;
else if(tank1.y<0)
tank2.y=0;
tank2.curr_frame=tank2 direction;
} } else {
j=-j;
tank1.curr_frame=tank1_direction*2+j;
tank2.curr_frame=tank2_direction+2+j; }
    Behind_Sprite((sprite_ptr)&tank1);
    Behind_Sprite((sprite_ptr)&tank2);
    if(bullet1.flag==1)
    bullet1.color_gd=Get_Pixel(bullet1.x,bullet1.y);
    if(bullet2.flag==1)
    bullet2.color_gd=Get_Pixel(bullet2.x,bullet2.y);
    Draw_Sprite((sprite_ptr)&tank1);
    Draw_Sprite((sprite_ptr)&tank2);
    if(bullet1.flag==1) Plot_Pixel_Fast(bullet1.x,bullet1.y,bullet1.color);
    if(bullet2.flag==1) Plot_Pixel_Fast(bullet2.x,bullet2.y,bullet2.color);
    if(Sprite_Collide((sprite_ptr)&tank1,(sprite_ptr)&tank2)||bullet_Collide((sprite_ptr)
&tank2,(bullet_ptr)&
    bullet1)||bullet_Collide((sprite_ptr)&tank1,(bullet_ptr)&bullet2)) {
    done=1;
    } Delay(2); }
    for(index=0;index<=300000;index++,Plot_Pixel_Fast(rand()%320,rand()%200,0));
    Set_Video_Mode(TEXT_MODE);
    Close_Serial(COM_1); }//关闭串口
```

14.4　本 章 小 结

键盘是游戏中最为常用的输入设备。如果能够自如地读取键盘数据我们就可以制作出高难度的用户操作规范。

清空键盘缓冲只需要将键盘缓冲尾指向键盘缓冲首指针。

鼠标是另一个常用的游戏输入设备。鼠标驱动程序在计算机中安装后可以被 BIOS 中断 33h 调用。而通过串口我们可以传输字符数据以及实现联机游戏编程。

学后建议

（1）制作一个检测多键同时按下和按顺序序列按下的游戏，比如一台机器上两个用户对打的游戏和街霸中同时按键发出的绝招密技；

（2）设计、制作一些你喜欢的鼠标形状，比如你的姓和名，用画笔绘制一个你喜欢的图像，然后将它在程序中调用成图像鼠标；

（3）制作一个两机对打的俄罗斯方块游戏，在游戏中一旦一台机器上用户消去 2 层以上的方块就要给另一台机器发出加方块层的数据信息。

第15章　界面技术

本章导读

经历过 DOS 到 Windows 飞跃的一代都会感叹：从最初的全键盘对文本到鼠标对图形，计算机世界发生了最大的一次“工业技术革命”。当我最初开始 C 语言图形编程的时候，便希望能够通过 C 语言仅仅模仿 Windows 的界面来造一个自己的“游戏 Windows 界面”，在那里也有任务栏、窗口、各类菜单和关闭按钮。不过要实现这样一个对象和对象事件繁多的界面谈何容易，在我以为要设计好的界面至少要掌握以下几方面的知识：

（1）自如驾驭屏幕二维图形和图像文件调用的能力；

（2）对结构体使用、链表知识、指向函数的指针有相当的认识；

（3）对 Windows 下面向对象编程有一定的理解，至少要理解对象、事件和事件触发等基本概念及其实际应用方法。

事实上，游戏中如果增加了界面，将大大提高交互性，方便玩者对游戏的操作。比如原先必须按某一个键来打开菜单，现在只需要用鼠标点击界面中的按钮就可以实现；此外，界面的应用还起到美化游戏场面的效果，使游戏看上去更加完整、亲切；更重要的是界面技术的使用将 C 语言游戏编程进一步带入了面向对象思路中去了。

事实上，界面技术其实完全可以扩展成整个游戏技术，学好这一章，就打下了游戏制作的深厚基础，这在本章的最后两节你就可以体会到。

本章将介绍如何设计界面对象的数据结构和 DOS 游戏下适合的界面方案。

本章重点

（1）面向对象事件触发思路在界面对象定义、对象事件函数制作和界面对象在程序触发中的应用；

（2）通过弹出菜单栏实现界面在 DOS 游戏中的实际应用，以及将界面插入以前制作的无菜单游戏可采用的多种方法。

15.1　界面对象的结构

15.1.1　对象的结构分析

Windows 中最常见的界面对象就是窗体、菜单和按钮，事实上在我们的游戏界面中也就这 3 个对象比较实用。那么这些界面对象拥有哪些主要的属性呢？见表 15-1。

表 15-1　窗体、菜单和按钮的属性对照表

对象	窗体	菜单	按钮
顶点	y	y	y
偏移	y	y	y
标题	y	y	y/n

（续表）

对象	窗体	菜单	按钮
文字颜色	y	y	y/n
背景颜色	y	y/n	y/n
当前形态	y	y	y/n
图标代表	y	n	y/n
热键	y/n	y	y
是否保存背景	y	y	y
当前是否有效	y	y	y

不难发现这三个界面对象拥有非常类似的结构，尤其是菜单和按钮的特点基本相同。事实上，和面向对象程序设计中构建对象类是差不多的，只是我们构建的结构里面没有函数而已。不过，可以通过在结构中设定指向函数的指针来完成。

下面我设计了一个简单的界面对象的结构体，除了考虑到以上一些共同属性和必须属性以外，还将增加每个对象的功能函数指针和在结构中实现了链表功能。

```
typedef struct point {
    int x;//x 坐标
    int y;//y 坐标
}point;
typedef struct windows{
    int kind;//界面对象类型
    point top;//界面对象坐标起点
    point move;//界面对象坐标偏移
    char *word;//名称、标注
    int color;//文字颜色
    int bk_color;//背景颜色
    int status;//界面对象当前形态状态
    char *hotkey;//热键
    void (far *windows)(struct windows *win);//指向界面对象事件函数的指针
    char far *background;//指向保存背景图像内存的指针
    int bk_flag;//是否要保存背景
    struct windows *next;//链表，指向下一个界面对象
    struct windows *father;//指向父指针（如弹出菜单选项）
    int active;//界面对象是否激活
} windows, *windows_ptr;
```

如果需要更多界面对象或者对象属性这个结构还需要扩充，当然，目前已经够用了。

15.1.2　对象的初始化

初始化对象函数如下：

```
void Object_Init(windows_ptr ptr, int kind, int x, int y, int move_x, int move_y, char
*word, int color, int bk_color, int status, char *hotkey, int bk_flag, void (far *windows), int
active){
        ptr->kind=kind;//初始化界面对象类型
        ptr->top.x=x; //初始化界面对象起始横坐标
        ptr->top.y=y; //初始化界面对象起始纵坐标
        ptr->move.x=move_x; //初始化界面对象偏移横坐标
        ptr->move.y=move_y; //初始化界面对象偏移纵坐标
        ptr->word=word; //初始化界面对象文字标注
        ptr->color=color; //初始化界面对象文字颜色
    ptr->bk_color=bk_color; //初始化界面对象背景颜色
    ptr->status=status; //初始化界面形态状态
    ptr->hotkey=hotkey; //初始化界面对象热键
    ptr->windows=windows; //初始化界面对象事件函数
    ptr->bk_flag=bk_flag; //初始化界面对象是否要保存背景
    if(bk_flag==1)//如果要保存背景，初始化界面对象背景图像内存空间
        ptr->background=(char far *)malloc( (move_x+1)* (move_y+1)+1);
    ptr->active=active; }//初始化界面对象的激活状态
```

初始化界面对象的过程如下：

（1）为当前对象的结构体申请空间；

（2）调用初始化函数。

程序实现如下：

```
//申请当前界面对象的内存空间
now=(struct windows *)malloc(sizeof(struct windows));
//初始化当前界面对象
Object_Init(now,WINDOWS,WINDOWS_X,WINDOWS_Y,WINDOWS_X_MOVE,
WINDOWS_Y_MOVE,"This is a game desktop",0,24,0,"",0,win);
```

以下是分别是窗体、关闭按钮和下拉菜单按钮的初始化：

1. 窗体

```
Object_Init(now,WINDOWS,WINDOWS_X,WINDOWS_Y,WINDOWS_X_MOVE,
WINDOWS_Y_MOVE,"This is a game desktop",0,24,0,"",0,win);
```

属性：使用 now 指针指向空间；建立一个窗体（WINDOWS）；x 起点为 WINDOWS_X；y 起点为 WINDOWS_Y；x 偏移为 WINDOWS_X_MOVE；y 偏移为 WINDOWS_Y_MOVE；

标题为“This is a game desktop”；前景色为黑色（0）；背景色为灰色（24）；不设状态；没有热键；无需保存背景；对应对象功能函数为 win()函数。

2. 关闭按钮

```
Object_Init(now,BUTTON_CLOSE,WINDOWS_X+WINDOWS_X_MOVE-BUTTON_XY-2,
WINDOWS_Y+2,BUTTON_XY-1,BUTTON_XY-1,"",0,24,0,"",0,clo);
```

属性：使用 now 指针指向空间；建立一个关闭按钮(BUTTON_CLOSE)；x 起点为 WINDOWS_X+WINDOWS_X_MOVE-BUTTON_XY-2；y 起点为 WINDOWS_Y+2；x 偏移为 WINDOWS_XY；y 偏移为 WINDOWS_XY；无文字标题；前景色为黑色（0）；背景色为灰色（24）；不设状态；没有热键；无需保存背景；对应对象功能函数为 clo()函数。

3. 下拉菜单按钮

```
Object_Init(now,BUTTON_MENU,WINDOWS_X+2,WINDOWS_Y+11,BUTTON_X-1,
BUTTON_Y-1,"Menu",0,24,MIN,"",0,men);
```

属性：使用 now 指针指向空间；建立一个菜单按钮(BUTTON_MENU)；x 起点为 WINDOWS_X+2；y 起点为 WINDOWS_Y+11；x 偏移为 WINDOWS_X_MOVE；y 偏移为 WINDOWS_Y_MOVE；标题为“Menu”；前景色为黑色（0）；背景色为灰色（24）；不设状态；没有热键；无需保存背景；对应对象功能函数为 men()函数。

如果界面对象在程序中没有用，可以使用对象删除函数来释放其中的空间，函数如下：

```
void Object_Delete(windows_ptr win){
    int index;
    if(win->bk_flag==1)//如果该界面对象保存背景，则首先释放其背景内存
        free(win->background);
    free(&win); }//释放该界面对象
```

15.1.3　界面设计与分析

界面程序设计的主要思路非常类似于 Windows 编程中的事件触发循环。只是在 DOS 下我们有机会面对更多难题。这给了我们一个整理思路的机会。

首先来看一下我们要面对多少问题：

（1）对象空间的申请；

（2）对象初始化；

（3）对象的绘制；

（4）对象事件函数的编写；

（5）鼠标的驱动；

（6）对象检测函数的编写；

这下死定了，许多在 Windows 中已经由编程平台和 Windows 内部实现的功能在这里需要我们自己来写。不过出于兴趣和研究角度，我们还是来尝试一下吧！

以下是我编写的程序大致流程（图 15-1）。

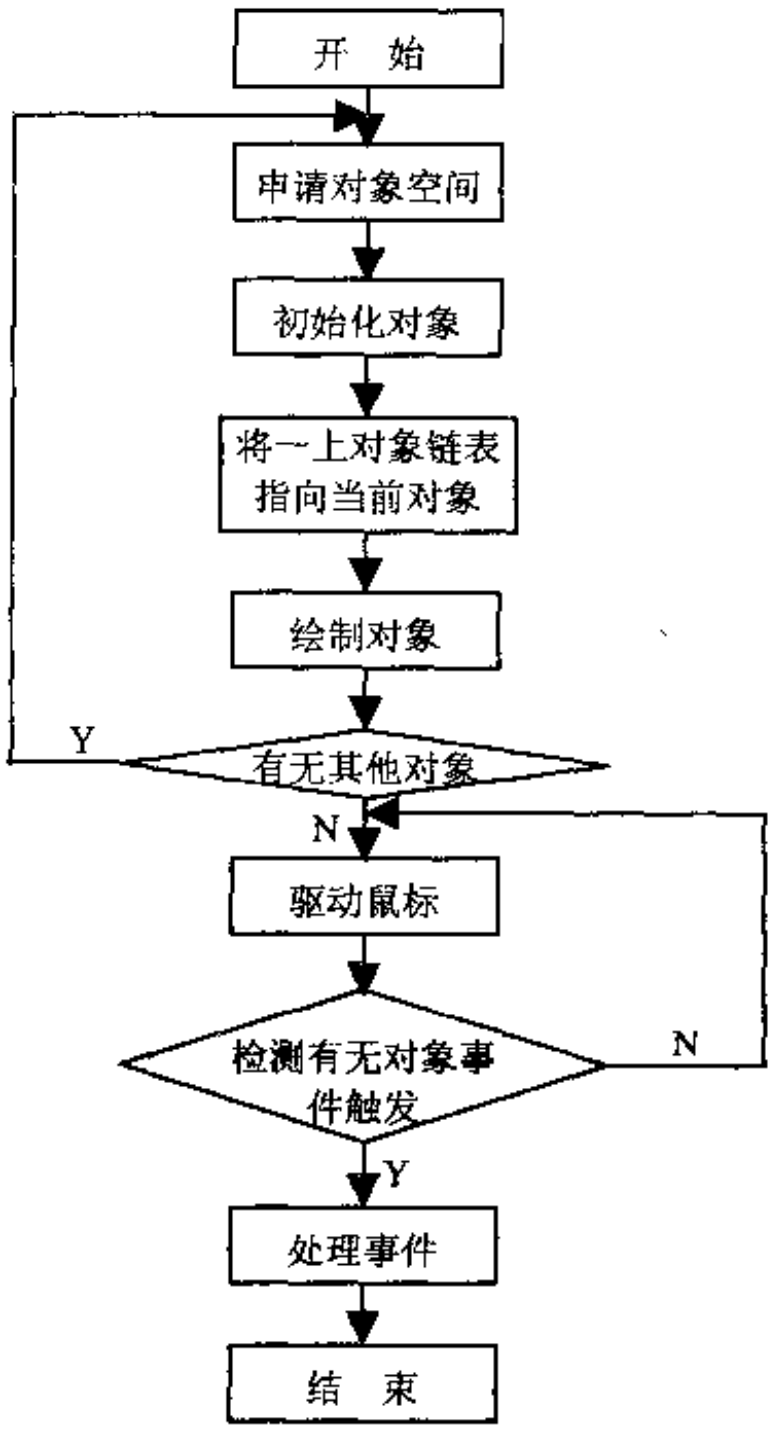


图 15-1　界面设计流程示意图

下面几节将对对象绘制、链表连接和事件检测进行逐一讨论，并实现一个较为完整的界面程序。

15.2　对象绘制函数

15.2.1　填充矩形绘制函数

对于窗体、按钮和菜单进行绘制事实上并不困难。可以使用第 6 章介绍的基本二维图形函数，这里用到最多的是绘制填充矩形的函数，虽然第 6 章没有介绍它的制作，但它其实是由横线函数派生出来的。

以下是填充矩形函数：

```
void Fill_Rectangle(int x1,int y1,int x2,int y2,int color){
    int i;
    for(i=y1;i<=y2;i++)     {
```

```
        H_Line(x1,x2,i,color);//横线函数
    } }
```

非常简单，其中的 H_Line()函数就是第 6 章介绍的横线绘制函数。而一个给定对角点（左上角和右下角）的填充矩形完全可以通过自上而下循环绘制横线来完成。当然有人问为什么不用自左向右的竖线函数 V_Line()来实现呢？道理很简单，看一下第 6 章的横线与竖线函数，横线函数是完整的内存复制而竖线函数是逐点绘制的，明显横线的绘制速度快了很多。

15.2.2　立体按钮绘制

有了填充矩形函数基本的对象形状就可以实现了。可是我们希望界面对象更加好看一点，比如 Windows 中立体按钮是如何绘制的？

如果仔细观察立体按钮，你会发现：按钮左边、上边有两条高亮的线；按钮右边、下边有两条加深的线；而按钮本身就是一个颜色介于两者之间的填充矩形。

于是，我们可以通过绘制三个填充矩形来完成一个立体按钮的绘制，比如要画一个从点(100,100)到点(130,120)的立体按钮，其过程如下：

（1）使用加深颜色绘制一个从左上顶点(100,100)到右下顶点(130,120)的矩形；

（2）使用高亮颜色绘制一个从左上顶点(100,100)到右下顶点(129,119)的矩形；

（3）使用按钮颜色绘制一个从左上顶点(101,101)到右下顶点(129,119)的矩形；

以下给出调用上一节填充矩形函数来实现立体按钮的代码：

```
Fill_Rectangle(100,100,130,120,8); //右下深色按钮边框
Fill_Rectangle(100,100,129,119,7); //左上浅色按钮边框
Fill_Rectangle(101,101,129,119,24); //按钮表面
```

由于按钮的颜色通常是灰色的，所以这里使用的三个灰色在调色板颜色寄存器的号码分别为：7（高亮），8（加深），24（按钮色）。其效果如图 15-2 所示。

当然，我们发现在左下顶点和右上顶点的高亮和加深颜色的接缝问题上，从光线原理上来说高亮和加深颜色应该是各自一半的。由于这里只有一个像素，这两个点使用的是加深颜色。也许在高分辨率下这算不了什么，但是在 320*200 的屏幕游戏分辨率下，立体效果会有所折扣。



图 15-2　立体按钮绘图

15.2.3　窗体、按钮和菜单绘制

界面对象的绘制其实就如同绘制一个立体的按钮，非常简单。我们想把窗体、按钮和菜单的绘制都放到一个绘制函数中来，使绘制过程变得有相当高的通用性。

仔细分析要绘制的对象情况，我将绘制类型分为：

（1）窗体：包括窗体主体、一个状态栏和菜单栏，标题在状态栏显示；

（2）菜单按钮：在菜单栏上的非立体的菜单按钮，有标题；

（3）按钮：普通突出的矩形立体按钮，有标题；

（4）按下的按钮：普通凹下的矩形立体按钮，有标题，是在被鼠标按下时的效果；

（5）关闭按钮：普通突出的正方形立体按钮，中间有一个大 X；

（6）按下的关闭按钮：普通凹下的正方形立体按钮，中间有一个大 X；

（7）最小化按钮：普通突出的正方形立体按钮，中间有一条小横线；

（8）按下的最小化按钮：普通凹下的正方形立体按钮，中间有一条小横线；

所有这八种对象的绘制共使用了 4 个图形、文字函数，它们分别是：

（1）横线绘制函数：

```
void H_Line(int x1,int x2,int y,unsigned int color)
```

（2）任意线绘制函数：

```
void Bline(int x0,int y0,int x1,int y1,unsigned char color)
```

（3）填充矩形绘制函数：

```
void Fill_Rectangle(int x1,int y1,int x2,int y2,int color)
```

（4）英文字符串显示函数：

```
void Blit_String(int x,int y,int color,char *string,int trans_flag)
```

以下是界面对象绘制函数：

```
void Draw_Windows(windows_ptr win){
switch(win->kind) {//判断当前界面对象的类型
    case WINDOWS: {//绘制窗体类型对象
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
move.y,WINDOWS_COLOR_DARK);
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,WINDOWS_COLOR_LIGHT);
    Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,WINDOWS_COLOR_OUT);
    Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->
top.y+10,WINDOWS_COLOR_TITLE);
    if(win->active==ACTIVE)Blit_String(win->top.x+2,win->top.y+2,win->color,win->word,1);
    else Blit_String(win->top.x+2,win->top.y+2,NOACTIVE_COLOR,win->word,1);
    Fill_Rectangle(win->top.x+2,win->top.y+22,win->top.x+win->move.x-2,win->top.y+win->
move.y-2,WINDOWS_COLOR_LIGHT);
    Fill_Rectangle(win->top.x+2,win->top.y+22,win->top.x+win->move.x-3,win->top.y+win->
move.y-3,WINDOWS_COLOR_DARK);
    Fill_Rectangle(win->top.x+3,win->top.y+23,win->top.x+win->move.x-3,win->top.y+win->
move.y-3,WINDOWS_COLOR_IN);
    break; }
        case BAR: {//绘制任务栏类型对象
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
```

```
move.y,WINDOWS_COLOR_DARK);
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,WINDOWS_COLOR_LIGHT);
    Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,WINDOWS COLOR_OUT);
    if(win->active==ACTIVE) {
        Blit_String(win->top.x+79,win >top.y+3,WINDOWS_COLOR_LIGHT,win->word,1);
        Blit_String(win->top.x+80,win->top.y+4,WINDOWS_COLOR_DARK,win->word,1);
    } else {
        Blit_String(win->top.x+79,win->top.y+3,WINDOWS_COLOR_LIGHT,win->word,1);
        Blit_String(win->top.x+80,win->top.y+4,WINDOWS_COLOR_DARK,win->word,1);
    } break; }
    case BUTTON_MENU: {//绘制菜单选项按钮类型对象
    if(win->active==ACTIVE)Blit_String(win->top.x+2,win->top.y+1,win->color,win->word,1);
        else Blit_String(win->top.x+2,win->top.y+1,NOACTIVE_COLOR,win->word,1);
    break; }
    case BUTTON: {//绘制普通按钮类型对象
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
move.y,BUTTON_COLOR_DARK);
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR_LIGHT);
    Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR);
//绘制按钮文字标注
    if(win->active==ACTIVE)//判断是否激活，不激活则用灰色绘制文字
        Blit_String(win->top.x+2,win->top.y+1,win->color,win->word,1);
        Else      Blit_String(win->top.x+2,win->top.y+1,NOACTIVE_COLOR,win->word,1);
        break; }
        case BUTTON_PRESS: {//绘制普通按钮被按下的情形
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
move.y,BUTTON_COLOR_LIGHT);
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR_DARK);
    Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR);
    Blit_String(win->top.x+2,win->top.y+1,win->color,win->word,1);
    break; }
    case BUTTON_CLOSE: {//绘制关闭窗体按钮类型对象
```

```
        Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
    move.y,BUTTON_COLOR_DARK);
        Fill_Rectangle(win->top.x,win->top.y,win->top.x-win->move.x-1,win->top.y+win->
    move.y-1,BUTTON_COLOR_LIGHT);
        Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
    move.y-1,BUTTON_COLOR);
        if(win->active==ACTIVE){
        Bline(win->top.x+2,win->top.y+2,win->top.x+win->move.x-2,win->top.y+win->move.y-2,
win->color);
        Bline(win->top.x+win->move.x-2,win->top.y+2,win->top.x+2,win->top.y+win->move.y-2,
win->color);
        } else {
        Bline(win->top.x+2,win->top.y+2,win->top.x+win->move.x-2,win->top.y+win->
    move.y-2,NOACTIVE_COLOR);
        Bline(win->top.x+win->move.x-2,win->top.y+2,win->top.x+2,win->top.y+win->
    move.y-2,NOACTIVE_COLOR);
    } break; }
        case BUTTON_CLOSE_PRESS: {//绘制关闭按钮被按下的情形
        Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
    move.y,BUTTON_COLOR_LIGHT);
        Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
    move.y-1,BUTTON_COLOR_DARK);
        Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
    move.y-1,BUTTON_COLOR);
        Bline(win->top.x+2,win->top.y+2,win->top.x+win->move.x-2,win->top.y+win->move.y-2,
win->color);
        Bline(win->top.x+win->move.x-2,win->top.y+2,win->top.x+2,win->top.y+win->move.y-2,
win->color);
        break; }
        case BUTTON_SMALL: {//绘制最小化按钮类型对象
        Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
move.y,BUTTON_COLOR_DARK);
        Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR_LIGHT);
        Fill_Rectangle(win->top.x+1,win->top.y+1,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR);
        if(win->active==ACTIVE)
        H_Line(win->top.x+2,win->top.x+win->move.x-2,win->top.y+win->move.y-2,win->color);
```

```
    Else H_Line(win->top.x+2,win->top.x+win->move.x-2,win->top.y+win->
        move.y-2,NOACTIVE_COLOR);
    break; }
    case BUTTON_SMALL_PRESS: {//绘制最小化按钮被按下状态
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x,win->top.y+win->
move.y,BUTTON_COLOR_LIGHT);
    Fill_Rectangle(win->top.x,win->top.y,win->top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR_DARK);
    Fill_Rectangle(win->top.x+1,win->top.y-1,win >top.x+win->move.x-1,win->top.y+win->
move.y-1,BUTTON_COLOR);
    H_Line(win->top.x+2,win->top.x+win->move.x-2,win->top.y+win->move.y-2,win->color);
    break; }
        default:break; } }
```

这里，为了提高对象绘制函数的通用性，我们做了--些常量定义：

```
#define WINDOWS 0 //绘制窗体
#define BUTTON 1 //绘制按钮
#define BUTTON_PRESS 2 //绘制按下的按钮
#define BUTTON_SMALL 3 //绘制最小化按钮
#define BUTTON_SMALL_PRESS 4 //绘制按下的最小化按钮
#define BUTTON_CLOSE 5 //绘制关闭按钮
#define BUTTON_CLOSE_PRESS 6 //绘制按下的关闭按钮
#define BUTTON_MENU 7 //绘制菜单按钮
#define BUTTON_COLOR 24 //按钮色
#define BUTTON_COLOR_LIGHT 7 //高亮色
#define BUTTON_COLOR_DARK 8 //加深色
#define BUTTON_X 35 //按钮宽度
#define BUTTON_Y 10 //按钮高度
#define BUTTON_XY 8 //正方形按钮边长
#define WINDOWS_X 50 //窗体 x 起始位置
#define WINDOWS_Y 50 //窗体 y 起始位置
#define WINDOWS_X_MOVE 240 //窗体 x 方向偏移
#define WINDOWS_Y_MOVE 150 //窗体 y 方向偏移
#define WINDOWS_COLOR_OUT 24 //窗体外部色
#define WINDOWS_COLOR_IN 15 //窗体内部色
#define WINDOWS_COLOR_LIGHT 7 //窗体高亮色
#define WINDOWS_COLOR_DARK 8 //窗体加深色
#define WINDOWS_COLOR_TITLE 12 //窗体状态栏颜色
#define ACTIVE 1 //对象无效
```

```
#define NOACTIVE 0 //对象有效
#define ACTIVE_COLOR 1 //有效颜色
#define NOACTIVE_COLOR 8 //无效颜色
```

如果我们在函数中不使用这些定义，那么要对对象进行修改就非常困难了。比如，如果要改变窗体的起始位置我们就需要修改绘制窗体部分的起始点和所有窗体内其他对象的起始点，而现在我们只需要将 WINDOWS_X 和 WINDOWS_Y 修改成新位置就可以了。

15.3　使 用 链 表

在界面程序中引入链表概念的目的是为了在对象事件检测函数中方便地循环检测所有对象是否被触发。

在界面对象结构体中我们设定了一个指向结构体的指针（struct windows *next;），用以实现界面对象链表。在每次初始化对象之后我们都将上一个对象的 next 指针指向当前对象，实现两个对象的链接。具体程序中链表头、链表中间部分和链表尾的操作过程有所不同。

以下给出链表链接的全过程：

（1）设定全局界面对象指针变量 head（指向链表头）、now（指向当前链表）和 pre（指向上一个链表）；

（2）在申请第一个对象空间的时候，将 head、now 和 pre 都指向它；

（3）在申请下一个对象空间的时候，将 now 指向它；

（4）将 pre->next(上一个界面对象的链表指针)指向 now；

（5）将对象指针 pre 指向对象指针 now（pre=now）；

（6）循环 3、4、5 步骤，直到申请完最后一个对象；

（7）将 pre->next 指向 NULL（最后一个链表指向空）。

以下给出程序实现过程。

全局变量设定：

```
//对应步骤 1
windows_ptr head;//指向界面对象链表头
windows_ptr now;//指向当前界面对象
windows_ptr pre;//指向上一个界面对象
```

主函数中初始化、绘制并且链接对象：

```
now=pre=(struct windows *)malloc(sizeof(struct windows));//对应步骤 2
Object_Init(now,WINDOWS,WINDOWS_X,WINDOWS_Y,WINDOWS_X_MOVE,
WINDOWS_Y_MOVE,"This is a game desktop",0,24,0,"",0,win,ACTIVE);
Draw_Windows(now);
head=now;//对应步骤 2
now=(struct windows *)malloc(sizeof(struct windows));//对应步骤 3
Object_Init(now,BUTTON_MENU,WINDOWS_X+2,WINDOWS_Y+11,BUTTON_X-1,BUTTON_Y-1,
```

```
"Menu", 0, 24, MIN, "", 0, men, ACTIVE);
Draw_Windows(now);
pre->next=now;//对应步骤 4
pre=now;      //对应步骤 5
now=(struct windows *)malloc(sizeof(struct windows));//对应步骤 3
Object_Init(now, BUTTON_SMALL, WINDOWS_X+WINDOWS_X_MOVE-(BUTTON_XY+2)*2,
WINDOWS_Y+2, BUTTON_XY-1, BUTTON_XY-1, "", 0, 24, MAX, "", 0, sma, NOACTIVE);
Draw_Windows(now);
pre->next=now;//对应步骤 4
pre=now;      //对应步骤 5
now=(struct windows *)malloc(sizeof(struct windows));//对应步骤 3
Object_Init(now, BUTTON_CLOSE, WINDOWS_X+WINDOWS_X_MOVE-BUTTON_XY-2,
WINDOWS_Y+2, BUTTON_XY-1, BUTTON_XY-1, "", 0, 24, 0, "", 0, clo, ACTIVE);
Draw_Windows(now);
pre->next=now;//对应步骤 4
pre=now;      //对应步骤 5
pre->next=NULL;//对应步骤 7
```

由于界面程序相对比较简单，并没有涉及到链表的插入和删除问题。只是在菜单下拉过程中从链表最后进行增加，以及在菜单关闭时有一个断链的过程。这将在下一节功能函数中讲到。

15.4　对象事件函数

所谓对象事件函数就是鼠标点击按钮或者菜单时产生的对应事件。比如，关闭按钮如果被触发，程序将退出；菜单按钮被触发，将弹出菜单。

所有这些界面对象的时间函数都是通过对鼠标的循环检测来触发的。这将在下一节中介绍。这里我们将介绍如何写界面对象事件函数。

15.4.1　按钮的基本动作

当按钮被按下的时候，其事件函数主要进行的操作包括：

（1）按下按扭图像；

（2）按钮恢复图像；

（3）该按钮触发的具体事件；

请注意 Windows 下的事件函数仅仅是对象被触发的具体事件；由于在 DOS 下，所以我们把按钮动画也放到事件函数中处理。

在窗体中比较常用的按钮是“关闭按钮”、“最小化按钮”和“最大化还原按钮”，而 DOS 游戏中“最小化按钮”和“最大化还原按钮”并没有什么实际意义。这里着重介绍的是关闭按钮。

以下是“关闭按钮”的事件函数：

```
void clo(windows_ptr win){
win->kind=BUTTON_CLOSE_PRESS.//设定界面对象类型为按下的关闭按钮
Draw_Windows(win);//绘制界面对象
Delay(5);//延迟一定时间
win->kind=BUTTON_CLOSE;// 设定界面对象类型为关闭按钮
Draw_Windows(win);//绘制界面对象
Delay(5);//延迟一定时间
Set_Video_Mode(TEXT_MODE);//恢复到文本模式
exit(1); }//退出程序
```

前 3 行通过调用上一节提到的界面对象绘制函数来实现按下按扭效果；
接下来的 3 行同样调用界面对象绘制函数来实现按钮恢复效果；
最后两行则是触发了关闭按钮的具体事件：设置文本显示模式和退出程序。

15.4.2　菜单的基本动作

菜单主要包括两种，一种是下拉菜单，一种是快捷菜单（鼠标右击）。由于实际制作方法类似，这里着重介绍的是下拉菜单。当菜单按钮被触发，其事件函数主要进行的操作和普通按钮基本相同：

（1）按下按扭图像；（可以省略）
（2）按钮恢复图像；（可以省略）
（3）菜单下拉；
（4）将下拉菜单中的对象加入对象链表；
（5）设置菜单为打开标示。

当第二次按菜单按钮的时候触发的事件为：

（1）按下按扭图像；（可以省略）
（2）按钮恢复图像；（可以省略）
（3）下拉菜单关闭；
（4）将下拉菜单中的对象从链表中删除；
（5）设置菜单为关闭标示。

这里需要说明的是，我把下拉菜单看作一组按钮群，比如，menu 菜单中有两个菜单选项 game 和 exit。当 menu 菜单按钮被触发的时候，我实际上制作了 game 和 exit 两个按钮。

事实上，相对于按钮按下的事件函数，菜单事件函数复杂了很多。主要是当其被触发的时候又套入了新的界面对象。以下给出一个简单菜单函数：

```
void men(windows_ptr win){
    windows *restore;
    if(win->status==MIN) {//判断菜单形态状态是否为最小化
    //构建菜单选项对象群
        now=(struct windows *)malloc(sizeof(struct windows));
    Object_Init(now,BUTTON,WINDOWS_X+4,WINDOWS_Y+BUTTON_Y+13,BUTTON_X-1,
```

```
BUTTON_Y-1,"game",0,24,0,"",1,bt1,ACTIVE);
    now->father=win;//菜单选项对象父界面对象为菜单对象
    Behind_Button(now);//保存菜单选项按钮背景
    Draw_Windows(now);//绘制菜单选项按钮
    pre->next=now;//将链表指向当前对象
    pre=now;
    now=(struct windows *)malloc(sizeof(struct windows));
Object_Init(now,BUTTON,WINDOWS_X+4,WINDOWS_Y+BUTTON_Y*2+13,BUTTON_X-1,
BUTTON_Y-1,"exit",0,24,0,"",1,bt2,ACTIVE);
    now->father=win;
    Behind_Button(now);
    Draw_Windows(now);
    pre->next=now;
    pre=now;
    pre->next=NULL;
    win->status=MAX;
} else if(win->status==MAX) {//菜单打开状态下按下菜单按钮释放所有对象(弹出菜单)
    restore=head;
    if(restore->bk_flag==1) {//恢复菜单选项按钮遮盖的背景内容
        Erase_Button(restore);//恢复背景
        Object_Delete(restore);//释放对象空间
    } do { restore=restore->next;//指向链表下一个对象
        if(restore->bk_flag==1)     {
            Erase_Button(restore);
            Object_Delete(restore); }
    }while(restore->next!=NULL);
    pre->next=NULL;
    win->status=MIN;
} }
```

程序中的 if 判断语句主要是区别当前菜单按钮是否处于关闭标示，如果是就做 if 里面的事情，否则将做 else 里面的事情。

在 if 里面保存了下拉菜单后的屏幕内容同时绘制和构建两个新的按钮（下拉菜单中的 game 和 exit），并且将它们的事件函数 btn1()和 btn2()也加入到对象链表，从而在以后的鼠标检测中能够触发这两个新的界面对象事件。最后设置菜单标示为打开状态。

在 else 里面将刚才打开菜单时菜单后的内容重新恢复到屏幕，并且将菜单中的按钮全部从链表中释放。最后设置菜单标示为关闭状态。

对于菜单后面屏幕内容的保存和子画面动画非常类似，以下给出保存界面对象后面内容的函数和恢复界面对象后面内容的函数。

保存：

```
void Behind_Button(windows_ptr win){
char far *work_back;
int work_offset=0,offset,y;
work_back = win->background;
//计算按钮对象背景在显存中的位置
offset = (win->top.y << 8) + (win->top.y << 6) + win->top.x;
//屏幕中按钮背景图像保存到该按钮背景内存
for (y=0; y<BUTTON_Y; y++){
_fmemmove((void far *)&work_back[work_offset],(void far *)&video_buffer[offset],BUTTON_X);
offset  += SCREEN_WIDTH;
work_offset += BUTTON_X; } }
```

恢复：

```
void Erase_Button(windows_ptr win) {
char far *work_back;
int work_offset=0,offset,y;
work_back = win->background;
//计算按钮对象在显存中的位置
offset = (win->top.y << 8) + (win->top.y << 6) + win->top.x;
//该按钮背景内存恢复到屏幕中按钮背景图像
for (y=0; y<BUTTON_Y; y++) {
_fmemmove((void far *)&video_buffer[offset],(void far *)&work_back[work_offset],BUTTON_X);
offset  += SCREEN_WIDTH;
work_offset += BUTTON_X; } }
```

在菜单事件函数触发后又新生了两个按钮对象，并有各自的事件函数 btn1()和 btn2()：

```
void bt1(windows_ptr win){
int x,y,fcolor;
long i;
win->kind=BUTTON_PRESS;//界面对象类型，按下的按钮
Draw_Windows(win);//绘制按钮
Delay(5);//延迟一定事件
win->kind=BUTTON;//界面对象类型，普通按钮
Draw_Windows(win);//绘制按钮
men(win->father);//调用父菜单事件函数，恢复弹出式按钮
for(i=0;i<1000000;i++) {//此按钮具体事件，随机产生一系列点
     x=rand()%(WINDOWS_X_MOVE-20);
     y=rand()%(WINDOWS_Y_MOVE-40);
```

```
        fcolor-rand()%(256)+1;
        Plot_Pixel_Fast(WINDOWS_X+10+x,WINDOWS_Y+30+y,fcolor); } }
void bt2(windows_ptr win) {//此按钮事件是退出程序
    win->kind=BUTTON_PRESS;
    Draw_Windows(win);
    Delay(5);
    win->kind=BUTTON;
    Draw_Windows(win);
    Delay(5);
    Set_Video_Mode(TEXT_MODE);//设置文本模式
    exit(1); }//退出程序
```

game 按钮的事件函数 btn1()除了绘制按下和恢复按钮图像动画，主要是在窗口中随机产生了许多点。我们完全可以将一个游戏写在这里替换它。

exit 按钮的事件函数 btn2()和前面介绍的关闭按钮事件函数 clo()的功能是完全相同的。

事实上我们可以根据需要在菜单事件函数 men()中加入任意多的按钮对象。当然你必须事先写好每个新按钮对象的事件函数。

15.5　进行事件检测

鼠标检测函数，实际上就是在所有界面对象链入链表以后，通过对鼠标位置和按钮的判断，来检测是否有界面对象被触发。如果有就进入该对象的事件函数中去，如果没有则继续检测。以下是一个事件检测函数例程：

```
void Judge_Object(void){
    int xmouse,ymouse,btnmouse;
    static int btn=-1;
    windows_ptr test;
    Squeeze_Mouse(MOUSE_BUTT_POS,&xmouse,&ymouse,&btnmouse);//检测鼠标状态
    if(btnmouse==btn)  return; //忽略鼠标重复的状态
    else btn=btnmouse;
    xmouse=xmouse/2;
    ymouse=ymouse;
    if(btnmouse==1) {//检测鼠标左键按下
    //循环检测界面对象是否被触发，通过对对象所处位置和鼠标坐标进行比较
        test=head;
        if((xmouse)=test->top.x&&xmouse<=test->top.x+test->move.x)&&
        (ymouse)=test->top.y&&ymouse<=test->top.y+test->move.y)) {
            Squeeze_Mouse(MOUSE_HIDE,0,0,0);//隐藏鼠标
            if(test->active==ACTIVE)//如果此按钮被激活
```

```
                test->windows(test);//触发此对象事件
            Squeeze_Mouse(MOUSE_SHOW,0,0,0);//显示鼠标
        } do {//继续检测链表中的对象
            test=test->next;
            if((xmouse>=test->top.x&&xmouse<=test->top.x+test->move.x)&&
            (ymouse>=test->top.y&&ymouse<=test->top.y+test->move.y)) {
                Squeeze_Mouse(MOUSE_HIDE,0,0,0);
                if(test->active==ACTIVE)
                    test->windows(test);
                Squeeze_Mouse(MOUSE_SHOW,0,0,0); }
        }while(test->next!=NULL); } }
```

程序首先通过 Squeeze_Mouse()函数来检测鼠标当前的状态，包括其位置和按键情况。我们感兴趣的当然是左键是否被触发了。

由于左键被触发后通常触发标示会持续相当一段时间，导致检测函数多次调用对应的对象事件函数。这里使用一个 if 语句来忽略重复的左键触发标示。

之后的 if 语句判断是否有鼠标左键被触发，如果有则将从界面对象链表头 head 开始检测鼠标位置是否落在其中任意一个界面对象中，如果落在其中则触发该对象的事件函数。检测直到所有界面对象都被检测过为止。

15.6　界 面 例 程

本节给出一个界面例程，它的界面对象包括：

（1）一个窗体（没有编写其具体事件函数）；

（2）一个关闭按钮（事件函数 clo()），以及最小化按钮（没有编写其具体事件函数）；

（3）一个菜单按钮（事件函数 men()）；

（4）菜单按钮触发后链入一个游戏按钮（用动画替代了游戏部分,事件函数 btn1()），一个退出按钮（事件函数 btn2()）；

程序功能：构建一个窗体界面，并且对其中的所有对象进行检测。

程序流程：

（1）初始化、并且绘制所有可见界面对象；

（2）驱动鼠标；

（3）驱动进入鼠标检测函数；

（4）处理检测到的对象事件函数；

（5）当检测到退出事件时，释放空间，推出程序。

主要函数：

```
void Draw_Windows(windows_ptr win); //绘制界面对象
void Object_Init(windows_ptr ptr,int kind,int x,int y,int move_x,int move_y,char
*word,int color,int bk_color,int status,char *hotkey,int bk_flag,void (far *windows),int
```

```
active); //初始化界面对象
    void Behind_Button(windows_ptr win); //保存子菜单按钮后内容
    void Erase_Button(windows_ptr win); //恢复子菜单按钮后内容
    void Object_Delete(windows_ptr win); //释放界面对象
    void win(windows_ptr win); //窗体事件函数
    void clo(windows_ptr win);      //关闭按钮事件函数
    void bt1(windows_ptr win); //子菜单 game 按钮事件函数
    void bt2(windows_ptr win); //子菜单 exit 按钮事件函数
    void sma(windows_ptr win); //最小化按钮事件函数
    void men(windows_ptr win); //start 开始菜单按钮事件函数
    void Judge_Object(void); //鼠标事件检测
```

程序要点：

所有界面对象都以链表串连，鼠标检测也是根据对象链表依次检测，从而在添加（菜单打开）和删除（菜单关闭）界面对象时依然保证所有对象被检测。其效果如图 15-3 所示。

程序代码 object.c 请查阅所附光盘的“source\15”目录。

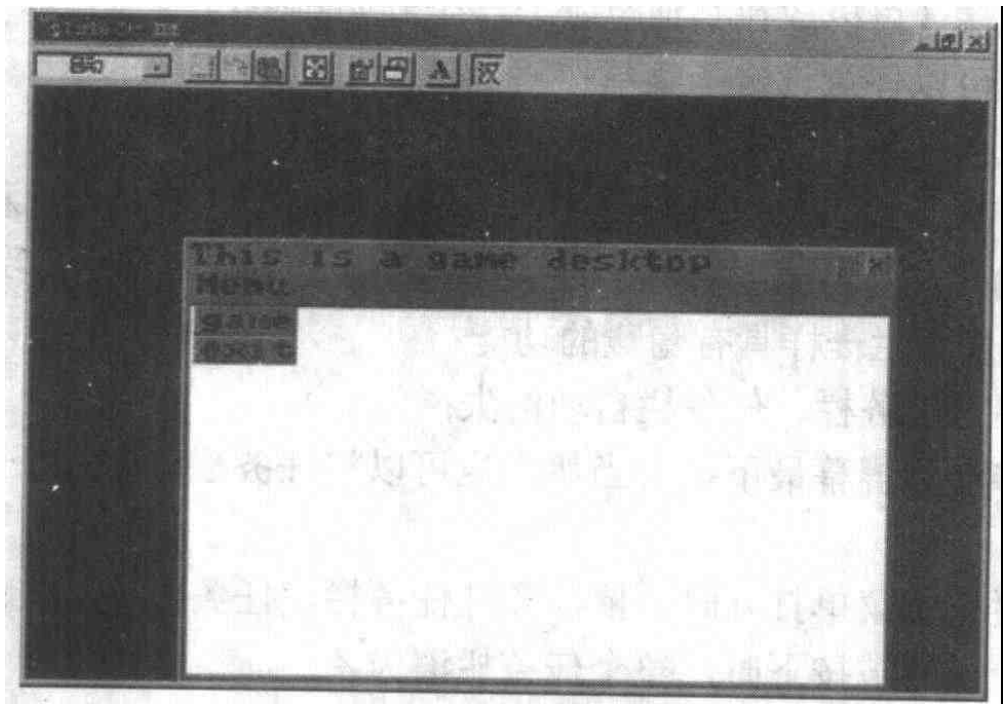


图 15-3　检测菜单事件

15.7　游 戏 实 例

15.7.1　DOS 游戏界面设计

我们这里采用 320*200 的较低分辨率来做 DOS 游戏，也就是说游戏最好是全屏幕的。如果采用窗体缩放，当缩小时由于分辨率更加低很可能使游戏中很多内容无法表现或者检测到。

事实上，如果采用窗体中的缩放功能对 DOS 编程的要求非常高，首先，所有对象的参照坐标点和对象大小都必须是一个全局缩放变量乘以其单位大小以便缩放重画使用；其次，必须对背景画面和各类子画面设计缩放函数。这对于面向过程语言来说是非常困难的。

此外，我们已经设计过许多全屏幕的 DOS 游戏，对于这些游戏如果要加入窗体界面也

是工作量极其大或者说没有实现可能的。

从 DOS 游戏界面作用来说，Windows 窗体的作用非常广泛：

（1）菜单作用；

（2）最大化，最小化，移动作用；

（3）关闭作用；

（4）标题说明作用；

（5）美化作用；

...

而 DOS 游戏界面的主要作用无非以下几点：

（1）关闭作用；

（2）菜单作用——开始、保存、读取、帮助和关闭；

（3）标题说明作用。

这个比较基础的 DOS 游戏界面不需要窗体，但需要菜单。根据 DOS 游戏界面 3 大作用分析，我们建立一个如同 Windows 任务栏的界面形式非常适合，将其作为 DOS 游戏任务栏。我构建的任务栏如下：

（1）任务栏左端（可以放在其他位置）一个开始菜单；

（2）开始菜单可以任意设定子菜单按钮数量；

（3）标题显示在任务栏上；

（4）任务栏右端（可以放在其他位置）提供最小化任务栏和退出程序快捷按钮。

考虑 DOS 游戏通常都是全屏幕的，而且要链入以前写的游戏中。我决定将这个任务栏做成如同 Windows 任务栏一样具有隐藏的功能：

（1）当鼠标离开任务栏，任务栏自动消失；

（2）当鼠标移动到屏幕最下端（当然你也可以将任务栏设定在任何位置），任务栏弹出；

（3）当任务栏开始菜单打开时，鼠标离开任务栏，任务栏不消失；

（4）当子菜单按钮被按下时，整个任务栏消失。

整个程序是根据上一章的界面例程修改而成的。

程序功能：构建一个通用的游戏任务栏界面，并且进行鼠标事件检测。

程序流程：

（1）初始化、并且绘制所有可见界面对象；

（2）驱动鼠标；

（3）驱动进入鼠标检测函数；

（4）处理检测到的对象事件函数；

（5）当检测到退出事件时，释放空间，推出程序。

主要函数：

```
void Behind_Object(windows_ptr win);//保存任意界面对象后的内容(包括任务栏条后面的内容)
void Erase_Object(windows_ptr win);//将界面对象后内容恢复显示
void Build_Object(); //构建和重新构建所有主界面对象
```

```
void Free_Object(void); //恢复所有对象后内容，并且释放所有对象
```

程序要点：

（1）任务栏隐藏和显示功能的实现依靠修改鼠标检测函数，在原有的检测鼠标按键前增加一个检测鼠标位置的语句，并且判断鼠标是否在屏幕最下方，如果在最下方（纵坐标 199）并且界面对象没有被构建则构建所有界面对象。判断鼠标是否离开了任务栏，如果离开了任务栏（纵坐标小于 186）并且界面对象处于构建状态同时菜单没有被打开则恢复有所对象后内容并释放所有界面对象。

（2）任务栏隐藏同时也可以由最小化按钮（隐藏任务栏按钮）实现。

（3）当点击 game 子菜单按钮后，所有子菜单按钮消失，同时任务栏也消失。任务栏被隐藏的原因是子菜单按钮释放后，鼠标处于任务栏上方，鼠标检测函数检测到后隐藏了任务栏。程序代码 gamebar.c 请查阅所附光盘的“source\15”目录。其效果如图 15-4 所示。

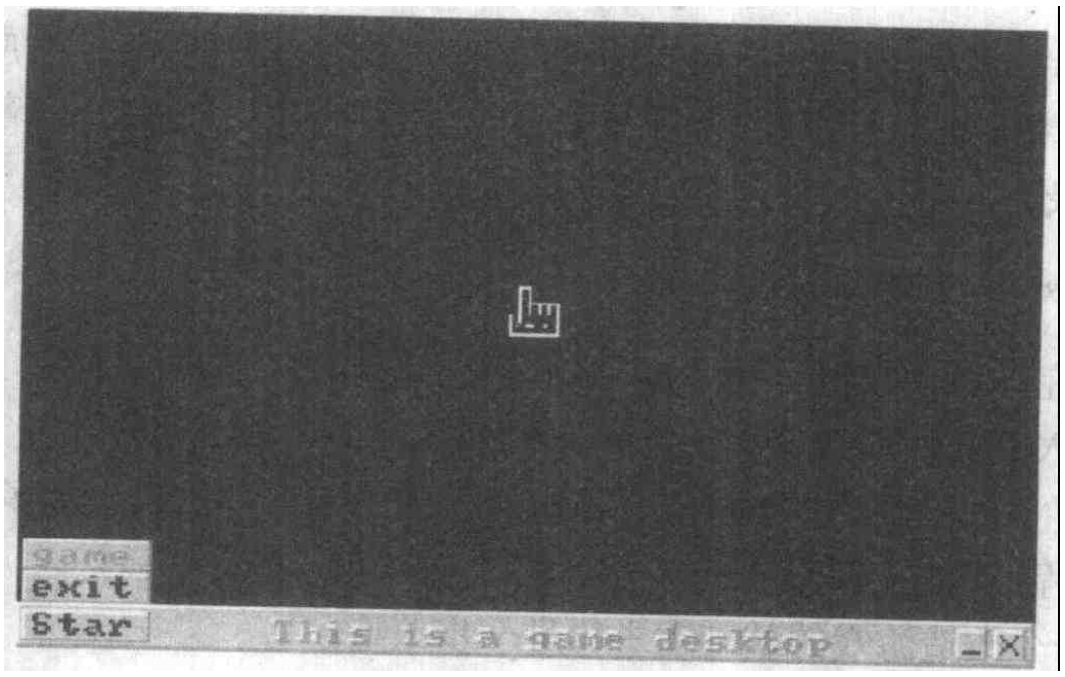


图 15-4　游戏界面

15.7.2　将界面插入游戏

1. 界面与游戏串行

程序功能：将第 11 章文件操作中移动矩形的例程加上游戏任务栏界面。开始(start，程序中用 Star 表示)菜单中包括“New”、“load”、“save”和“exit”子菜单按钮。

但是在任务栏按钮驱动游戏后，任务栏无法打开，直到按“q”键退出游戏以后任务栏才能重新驱动。

这种界面插入方式程序改造最方便，但使用起来较为麻烦。界面和游戏属于串行工作。

游戏改造过程：

（1）以上一节中的任务栏程序作为基础，将第 11 章移动矩形的例程插入其中；

（2）增加全局变量 xa,ya；

（3）菜单函数的修改；

a. 本程序使用到 4 个子菜单按钮，修改 men()函数，增加足够的按钮对象；

b. new 按钮对应的函数为 bt1()，在其最后调用初始位置变量设置和移动矩形游戏函数 game()，同时将此函数加在 bt1()函数前面；

c. exit 按钮没有发生变化；

d. load 按钮对应的函数为 loa()，此函数参照 bt1()函数，最后调用进度文件读取函数 Read_From_File()、初始位置变量赋值和 game()，同时将进度文件读取函数加入；

e. save 按钮对应的函数为 sav()，此函数参照 bt1()函数，最后调用当前位置变量赋值和进度文件读取函数 Write_To_File()，同时将进度文件写入函数加入；

主要函数：

```
void game(void); //移动矩形游戏
void Read_From_File(char flag,FILE *stream,int *word); //读进度文件
void Write_To_File(char flag,FILE *stream,int *word); //写进度文件
void loa(windows_ptr win); //读取进度按钮事件函数
void sav(windows_ptr win); //进度保存按钮事件函数
```

程序要点：

采用串行方式，将游戏加入通用界面程序时，只需要增加和修改对应的子菜单事件函数和子菜单对象就可以了。其它部分无须修改。程序代码 game1.c 请查阅所附光盘的“source\15”目录。

2. 界面与游戏并行

程序功能：在任务栏按钮驱动游戏后，任务栏同样可以触发。

这种界面插入方式程序改造较上一种比较麻烦，但游戏玩起来比较方便。界面和游戏算作并行（当然在 DOS 下面是不可能真正有并行的）工作，可以看似多任务处理。

游戏改造过程：

（1）在上一个改造游戏的基础上进行进一步改造；

（2）增加全局变量 gameflag（游戏当前是否开通状态标志）；

（3）game()函数的修改：game()移动矩形函数，原先是使用 while()循环形式来进行的（大多数游戏中都是如此）。但这里为了让其和任务栏并行工作，将 while()循环改为 if()判断语句；

（4）main()函数的修改：在任务栏主程序 main()的 while 循环中（本来只有鼠标检测函数）加入 game()游戏函数（在游戏状态标志开通的情况下调用）；

（5）菜单函数的修改；

a. 本程序使用到 4 个子菜单按钮，修改 men()函数，增加足够的按钮对象；

b. new 按钮对应的函数为 bt1()，在其最后调用初始位置变量设置和设置游戏状态标志为开通（gameflag=0）；

c. exit 按钮没有发生变化；

d. load 按钮对应的函数为 loa()，此函数参照 bt1()函数，最后调用进度文件读取函数 Read_From_File()、初始位置变量赋值和设置游戏状态标志为开通（gameflag=0）；

e. save 按钮对应的函数为 sav()，此函数参照 bt1()函数，最后调用当前位置变量赋值和进度文件读取函数 Write_To_File()；

（6）鼠标检测函数的修改：

只在判断到重建任务栏时将游戏状态表示关闭（gameflag=1）；

主要函数：无新的函数

程序要点：

（1）这里所谓的并行多任务效果只是说在游戏开始后任务栏还是能够被检测到的，但是当任务栏出现后游戏函数就没中止了；

（2）实现并行的关键是将所有要参与并行的函数都放入主函数的 while()循环中，而每个参与并行的函数自身不能是一个 while()循环函数。对于以前设计的游戏一定会有一个游戏函数 while()变 if()的修改过程。程序代码 game2.c 请查阅所附光盘的“source\15”目录。

3. 鼠标同时检测游戏和界面

程序功能：游戏和任务栏都是由鼠标控制的（而不是键盘控制），进行游戏时任务栏也可以触发。

游戏改造过程：

（1）在上一个程序基础上进行改造；

（2）game()函数的修改：game()函数加入鼠标横纵坐标参数，同时 if 循环原本是对按键进行判断的，现在改成对鼠标位置进行判断，如果在两个坐标方向上鼠标位置大于矩形当前位置则在该方向上移动；

（3）主函数的修改：从主程序 main()的 while 循环中去除 game()函数，还剩下一个鼠标检测函数；

（4）菜单函数不进行修改；

（5）鼠标检测函数的修改：在判断到鼠标按键的 if()语句的最后加入 game()函数（在 gameflag 为 1 的情况下开通）。

主要函数：无新函数

程序要点：

（1）戏中所有内容（界面对象和游戏对象）都由鼠标驱动，这实际上又一次使用到了事件触发思路。在界面对象的事件检测中我们采用的是链表方式对所有对象进行检测，然而由于是改写游戏，只能将游戏函数放在链表检测后被检测（可以被看作链表中的一个元素）。

（2）如果是设计新的游戏，建议将游戏中所有对象也如同界面对象一样建立一个统一的结构，并且链入链表，这样就不分界面对象还是游戏对象统统都被依次检测。

（3）这个程序给我们一个启示是，界面对象其实就可以完全扩展成游戏对象，游戏设计的思路和界面程序的设计是基本相同的。只要我们建立一个更强大的界面结构（游戏对象结构），整个游戏都可以用界面程序来改写出来。程序代码 game3.c 请查阅所附光盘的“source\15”目录。其运行结果如图 15-5 所示。

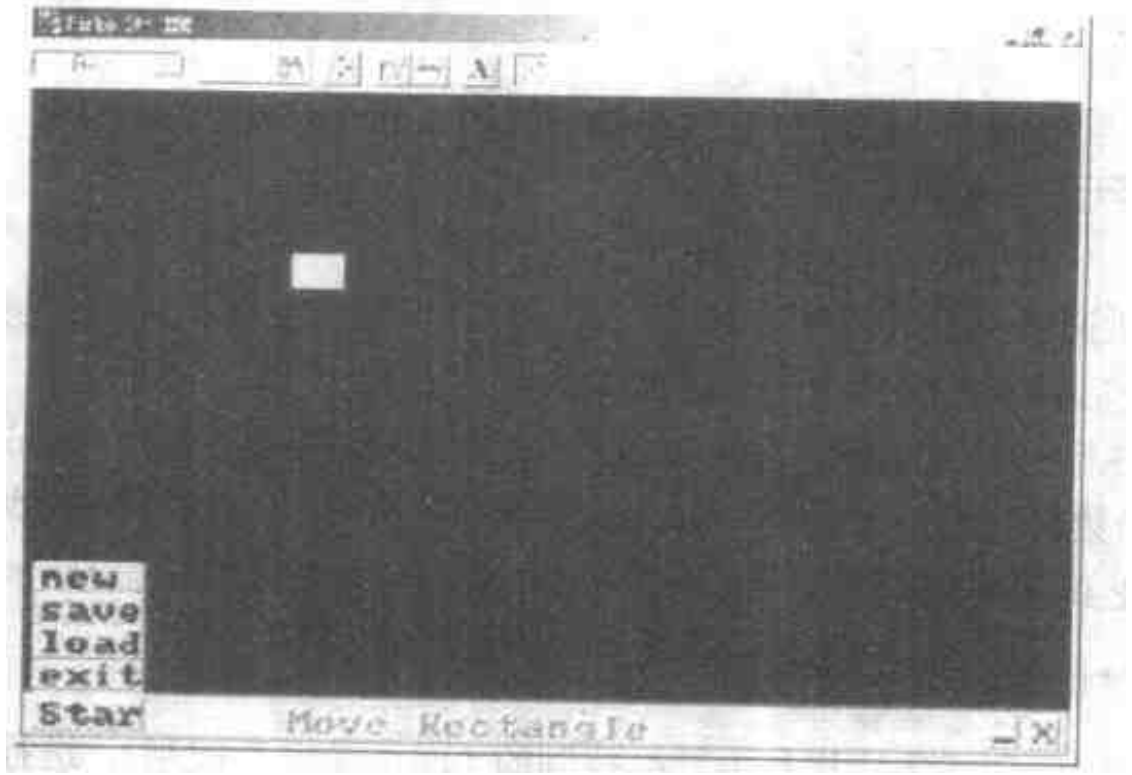


图 15-5　加入菜单及游戏对象后的界面

15.7.3　构建个性化界面

前面给出的那个任务栏通用界面非常好用，如果你有兴趣可以根据前面所讲的界面设计方法设计更加适合自己游戏的界面，可以做得更加有个性一些。

（1）使用基本图形函数绘制一些形状更加奇怪的任务栏或者按钮

a. 可以使用圆形绘制和填充函数设计圆形的按钮；

b. 注意检测方法将涉及判断是否在圆形内部的数学计算，当然你也可以使用矩形粗略判断；

c. 也可以使用多边形函数设计任意形状节面对象；

d. 还可以使用 pcx 图形显示函数和子画面技术调用 pcx 图片来给界面对象增加漂亮的“皮肤”；

（2）可以增加鼠标右键快捷菜单功能

这和普通菜单制作大同小异，不同之处在于是鼠标右键触发。

a. 菜单的参照坐标就是鼠标点击右键时鼠标坐标；

b. 绘制时同样要保存屏幕后内容。

（3）可以增加屏幕图标，使界面更加贴近使用者习惯

a. Windows 界面除了任务栏还有一些如同“我的电脑”这样的图标；

b. DOS 游戏界面中完全可以引入同样的概念，同样通过调用 ico 文件显示函数来制作；

c. 图标可以是你写的多个游戏，游戏者可以通过鼠标点击任意选择其中一个游戏玩。

（4）增加对话框功能

a. 在一些类似于 RPG 游戏中需要输入玩者的名字或者选择保存名的时候完全可以使用对话框；

b. 对话框的绘制和窗体的制作基本相同，完全可以使用界面对象结构；

c. 对话框中将输入内容放到对象结构表示对象标题的字符串变量 word 中；

d. 字符串 word 可以被游戏使用到或者可以作为游戏保存名。

DOS 游戏虽然无法如同 Windows 面向对象、可视化编程方便地实现界面设计，但给了我们一个机会去了解这些界面对象的结构和设计组和方法；同时它的底层化特点更能发挥自己的想象力，并将其变成现实。

15.8　本 章 小 结

方便用户操作是游戏制作中的一个要点，我们可以通过界面的制作来加以实现。在界面技术中采用面向对象的思路将大大提高程序效率。面向对象思路的在界面技术中的体现：

（1）界面对象的结构定义，可以使用链表指针、指向对象事件函数的指针；

（2）界面程序中对象空间的申请和释放，依次对界面对象链表检测，触发对应界面对象的事件函数。

在完成界面对象的初始化后，我们要面对的最大问题就是鼠标事件检测的函数。本章建议使用弹出任务栏界面对象，因此在事件检测函数中除了要通过链表依次检测是否有界面对象事件函数被触发外，还要判断鼠标是否处于界面的底端，如果处于底端则构建界面对象，如果不是在底端则释放界面对象空间。事实上，如果游戏中也使用到鼠标，我们在鼠标事件检测函数中还需要检测游戏事件函数是否被触发（当然你愿意花功夫也可以将游戏作为一个对象链入界面对象中，这样检测起来就非常简单了）。

学后建议

（1）制作你喜欢的个性化界面对象，进一步考虑在 DOS 游戏中如何将弹出任务栏改进的更加合理；

（2）尝试通过调用图像文件来制作界面对象中的各类图像标签和直接用按钮图像替代我们绘制的按钮对象；

（3）研究开发出窗体中最大化、恢复、最小化、右键菜单和窗体移动等对象的事件函数。

第16章 其他问题

本章导读

游戏编程其实还有许多问题可以探讨。比如中断、多任务、内存驻留；病毒、加密；优化技术和人工智能；面向对象思路在过程性语言中的使用，游戏数据库构建思路，各类游戏编程实用办法和通用游戏编程思路；更加深入的游戏编程，还有三维游戏编程等问题。

本章将着重介绍中断 TSR 驻留技术、简单的 C 语言文件感染病毒、面向对象思路的应用和分类游戏制作方法。虽然 TSR 技术在游戏制作中应用并不是很多，但是通过对驻留技术的学习，我们可以进一步地了解中断，事实上我们还可以通过中断在 DOS 下模仿多任务。C 语言文件病毒和游戏制作也不是非常关联，但是大多数喜欢写游戏的人通常对病毒制作都更加感兴趣，当然无超级公害的病毒在游戏软件保护方面还是有一定作用的。

本章重点

（1）TSR 驻留的原理、实现函数和具体实现例程；
（2）感染 C 语言文件的病毒实现原理及其 C 语言实现程序；
（3）面向对象思路在 C 语言中的使用及其好处；
（4）C 语言制作游戏分类及其编程思路、制作重点和难点。

16.1 TSR 驻留

如何让我们的游戏更加有趣，TSR 驻留程序就可以使游戏增色不少。我们只需要做一个批处理文件，让一些有目的性的驻留程序在我们的游戏前面运行。

16.1.1 TSR 基本知识

TSR 程序就是终止并常驻内存程序（Terminate and Stay Residemt）的简称。它是为了帮助在单用户、单任务 DOS 环境中加入多任务的成分。

要了解 TSR 首先必须熟悉中断和中断改写等一些概念。所谓中断是指 CPU 在正常运行一个程序时，由于程序中的事先安排或由外界事件的触发，导致 CPU 中断当前正在运行的程序，而转入相应的服务程序中去的过程。这些引起程序中断的事件称为中断源，程序安排的事件是指中断指令，程序执行到中断指令后，立即跳转到相应的服务程序中。实质上，CPU 在响应中断时，是按照中断源所对应的地址，引导程序跳转到相应的服务程序中的，这个与中断源有着一一对应关系的地址称为中断向量。

PC 机有两种类型的中断，即软中断和硬中断。软中断由执行某些指令产生。硬中断则是由接口设备引起。

PC 机在其内部存储了 256 个中断向量（详见附录 D），每个占用 4 个字节。每个中断向量用其类型码加以区别。实际执行过程中，CPU 根据其类型码，将其乘以 4 得到中断向量地址，即服务程序的入口地址。

我们在第 5 章图形模式中介绍过如何利用 int86()函数调用 BIOS 中断 10H 的 13H 功能来设置图形模式。除了可以调用各类中断以外，我们还可以改写中断。具体的方法是：

（1）保存所要改写的中断向量入口；

（2）将中断向量入口指向新的中断服务函数；

（3）在程序结束后恢复中断向量入口。

有关中断改写中断服务程序的函数请查阅附录 D。

TSR 程序必须监控一个或多个中断，并提供新的适应 TSR 程序的中断服务程序（如中断 08H，09H，28H，2FH）来完成相应的功能。

具体就是，TSR 程序是通过中断来触发的，这些中断主要包括键盘中断、时钟中断等。首先我们必须改写这些中断，将我们要求驻留程序触发具体触发方式和触发后的行为写入其中。接着和中断改写程序不同的是，我们在程序结束时并不恢复原先的中断向量入口，而是使用一个驻留函数 keep()将这个程序放在内存中。也就是说虽然程序运行结束了，但是程序并没有在结束时清除，只是处于休眠状态，待一个被实现设定好的触发条件（如同王子亲吻长眠公主的魔咒解数）被执行，程序就从内存中恢复了它的生机。关键是中断在计算机开机后的任意时刻都是发挥作用的（如同时钟中断始终在发生而键盘中断当你每次敲击键盘的时候都会触发），于是我们只是改写了中断做的事情（比如让时钟中断除了做其自身的事情以外再帮助我们显示当前时间、让键盘中断除了接收按键以外还能因为某些特殊按键启动一个菜单）。许多病毒就是用这种方法来发生其恐怖任务的（比如黑色星期五、CIH 等）。

有关 C 语言函数库提供的 TSR 函数请查阅附录 D。

驻留程序通常的制作过程：

（1）改写中断函数；

（2）使用 keep()函数将当前程序驻留在内存空间中。

16.1.2　时钟驻留

下面是利用 TSR 技术在 DOS 下建立一个时钟的程序。它利用定时器中断 1CH，由于该中断每秒钟产生 18.2 次，于是每 18 次中断修改一次显示时间。同时利用直接写屏技术将时间写到屏幕上。源代码 clock.c 请查阅所附光盘的“source\16”目录。

16.1.3　热键驻留

热建驻留是通过改写键盘中断 09H 来实现的。改写后的键盘中断程序首先调用原先的键盘中断服务程序，然后判断是否有热键被触发，一旦有热键触发则进行相应的操作。

通过功能键和普通键触发来实现屏幕反色闪动和屏幕恢复正常的功能的 TSR 例程 hotkey.c 请查阅所附光盘的“source\16”目录。

16.2　简 单 病 毒

对于病毒我们应该是深恶痛绝的，但是作为纯研究许多人还是很有兴趣的。游戏很难和病毒联系起来，不过写游戏的人通常会对研究病毒也有兴趣。比如我就喜欢养病毒，还

有五、六本病毒的书。

不过对于写病毒我并不在行，并且出版规定不可以公开病毒完整代码。于是这里我们将简单介绍一下病毒的编制原理并给出一个毫无伤害性的 C 语言伪病毒来说明问题。一切都只是出于编程研究。

病毒的最大特点就是自我复制。从病毒的分类来说有很多种，这里我们将介绍最流行的附加式病毒。它通过对正常的文件进行改写、增加来实现其自我复制的目的。

从程序角度来说，我们要做的事情有两件：

（1）让程序能够将自己在不影响其它程序本身工作的情况下复制给其他程序，使它具备继续复制的能力；

（2）在一定条件下使其产生某种发作效果。

第一件事情实际上可以看成对文件进行复制，把病毒源文件的功能函数全部放到被感染文件的最后，同时在被感染文件中调用这个函数。以下给出 C 语言的实现过程：

（1）主程序调用病毒功能函数；

（2）病毒功能函数读取查找同目录下所有 C 文件；

（3）找到一个（被感染 C 文件），打开它，并且将此文件全部读取到数组变量；

（4）重新创建一个同名文件（被感染 C 文件）；

（5）数组变量写回这个被感染 C 文件，同时将病毒源文件所需要的头文件、病毒功能函数调用语句写入；

（6）打开病毒源文件，将病毒功能函数全部写到被感染 C 文件的最后。

一个简单的 C 语言伪病毒 virus.c 请查阅所附光盘的“source\16”目录，virus.c 运行后其内容变化另保存为 after_virus.c。

此时，如果我们将 1.c 文件用 A 盘复制到其他机器或者 Email 给同学，结果它们一运行又感染了它们保存 1.c 文件目录下所有 C 文件。

对于第二件事情——“发作效果”，这里只用 printf 语句警告了一下，当然你完全可以写一个 TSR 驻留函数。

16.3　OOP 应用

过程性语言和面向对象语言 OOP 应该属于两大编程流派，不过面向对象语言发源于过程性语言。面向对象语言的产生实际是过程性语言在大规模（数千万句）编程中程序实现效率极低、出错率极高和维护、修改、扩展及其困难的基础上发展而来的。同时面向对象语言也是在过程化语言不断发展、总结的基础上实现的。当 C 语言中用结构体描述对象的时候其实已经有了面向对象思路的萌芽，只是它比类少了成员函数（只是从某种角度说）。

过程性语言已经走向末路，我们之所以用 C 语言来制作游戏同时要加入面向对象思路也只是希望打下较为坚实的游戏编程基础和为今后的面向对象游戏设计做铺垫。于是在程序设计中尽可能多地融入面向对象思路对我们很有帮助。事实上 C 游戏编程中更多加入面向对象思路也会使我们的游戏设计更加容易修改扩充和维护，编程效率也将远远高于从前。

C 语言其实也给了我们这种可能，比如我们可以在结构体中使用指向函数的指针来替代成员函数，让被指向的函数来完成对象事件；此外，在结构体描述对象的时候增加链表

来为事件触发循环作铺垫。OOP 游戏设计结构如图 16-1 所示。

面向对象思路设计游戏的结构图如下：

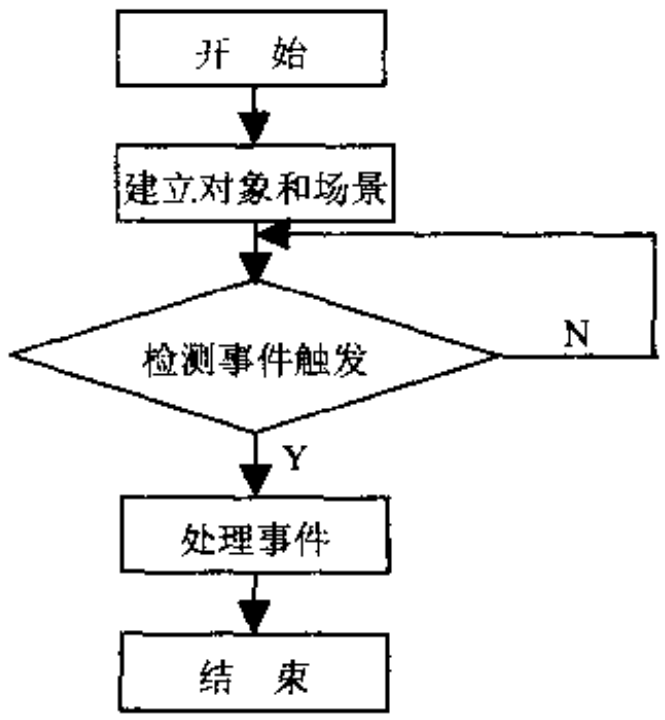


图 16-1　OOP 游戏设计流程示意图

面向对象的思路在第 10 章子画面技术和第 15 章界面技术中都应用到了，其中主要表现为：

（1）对象的结构体中增加了事件函数指针和对象链表；

（2）在游戏前进行对象构建；

（3）游戏主循环通过检测对象事件触发条件（通常是鼠标和键盘）来进行，一旦有事件被触发就去调用该对象的事件函数；

（4）与没有用的对象进行释放；加入了面向对象思路的 C 游戏编程到底有如何的效率提高，这一点我在写第 10 章中深有体会。从前子画面在主循环中都要对每个方向按键进行判断，然后做出相应的计算；而现在只需要在构建子画面对象时增加字符型按键数组变量，在初始化子画面对象时将所需要按键放入，在游戏运行时只需要比较按键和子画面按键数组就可以了。这样一来如果要修改具体的方向按键，只需要在构建语句中改一下按键字符就可以了，而不需要像从前到主循环中去找到每个按键字符去修改了。

但是在第 10 章中我们所谓的事件触发还带有强烈的过程性特点，或者说根本还不能算事件触发。原因在于，我们虽然在对象构建中有控制键，但是在游戏触发主循环中我们只是循环调用对象事件函数，所有控制键的读取和检测都是在事件函数中完成的。从事件触发思路来讲，我们应该在游戏主循环中检测键盘按键，然后和每一个对象的控制键去比较，一旦发现相同的则触发这个控制键的事件函数。也就是说键盘读取、控制键检测这两件事情应该放在事件函数外面来处理。在子画面对象程序设计中这一点是应该改进的。

不过到了第 15 章界面技术的时候，事件触发的思路就比较清晰了。我们在游戏主循环中对鼠标进行循环检测，一旦鼠标被按下，就通过链表循环检测鼠标是否位于界面对象的区域。如果是则调用该界面对象的事件函数。

以下是 C 语言游戏编程面向对象思路的进一步扩展建议：

（1）可以将游戏中所有对象同等看待，比如界面对象和游戏子画面和其他对象都可以作为同一概念上的对象来处理，从一个结构体（如同类）出发，在其基础上再设计自身的

结构体（如同子类），所有的对象结构体都有链表指针、事件函数指针和触发条件变量（比如针对键盘的控制键，针对鼠标的位置区域）；

（2）将所有能够触发对象事件的触发因素都统一放入游戏主循环的对象事件检测中，比如将鼠标检测和键盘检测一起放入检测循环，一旦检测到有鼠标或者键盘动作，则继续在事件检测循环中通过对链表每个对象触发条件匹配来判断此动作是否触发了对象事件；

（3）为鼠标和键盘开辟一定的缓冲空间，以保证在一次对象轮寻中没有发生遗漏触发现象。

当你希望将所有游戏中的内容（包括各类文件操作、背景画面）都做成对象的时候，建议干脆使用真正的面向对象语言吧！

16.4　各类游戏编程思路

除了将面向对象思路融入 C 语言游戏编程以外，每种类型的游戏也还是有自己的独特设计思路的。游戏类型主要包括：

（1）桌面游戏；

（2）视频战斗游戏；

（3）魂斗罗类游戏；

（4）玛丽、赛车类游戏；

（5）RPG 游戏。

16.4.1　桌面游戏编程思路

桌面游戏举例：俄罗斯方块、扫雷、大富翁。

桌面游戏的制作重点：

（1）实现游戏的算法；

（2）界面、图像制作的精巧。

桌面游戏的制作难点：

此类游戏的形式非常多样，不过在屏幕图像实现和用户输入方面基本没有什么特殊要求和难点。主要表现为每个游戏本身逻辑结构设计方面的实现问题上。比如俄罗斯方块游戏的数据结构描述问题；扫雷游戏的“类填充问题”（点到非地雷时候周围扩散的算法）；大富翁游戏中大量游戏数据存储、变化和相互关系问题。

桌面游戏的制作思路要点：

（1）根据游戏要求设计良好的数据结构；

（2）制作精致的游戏多媒体场景；

（3）在游戏编制过程中根据数据结构设计出简便的算法。

桌面游戏的制作方案举例（如图 16-2）：

游戏名：俄罗斯方块

我曾经用图形模式和文本模式制作过俄罗斯方块游戏。文本模式下将“1”作为基本方块单元，在文本模式下俄罗斯方块的数据结构可以直接呈现在屏幕上。

游戏规则：

（1）几种形式方块下落，允许转动，移动；

（2）从游戏槽的底端向上堆积，如果出现同一层全部是方块则消去该层，计算出消去层数；

（3）当游戏槽被堆满后，游戏结束。

游戏数据结构设计：

（1）将组成每个方块和游戏槽的正方形最小单元作为最基本的数组单元，称为基本方块；

（2）使用二维数组描述每个方块的基本形态，数组取值 1 表示该位置有基本方块；

（3）使用二维数组描述游戏槽，数组取值 0 表示该位置空，1 表示该位置存在方块。

游戏部分算法实现：

（1）方块转动问题：可以在设计方块形态时全部初始化，也可以根据方块基本形态进行矩阵变换获得；

（2）层消去问题：每次下落完成后调用检测函数，对游戏槽数组的所有行查询，如果出现一行都是 1 的情况则将所有后面的数组行下移，数组最后一行清 0，并且消去层数加 1；

（3）下落停止判断问题：每次方块下落一层前判断方块数组（从游戏槽数组对应位置取出）每列的最低端游戏槽数组单元是否为 1，只要出现为 1 情况就停止；

（4）变形问题：当下落方块边上有已经堆好的方块时，有时候是无法进行变换的，此时可以采用类似于下落停止判断的方法，在用户按变形键后检测变换后形状所占用（为 1）的游戏槽数组是否已经有为 1 的，如果有则无法变换。

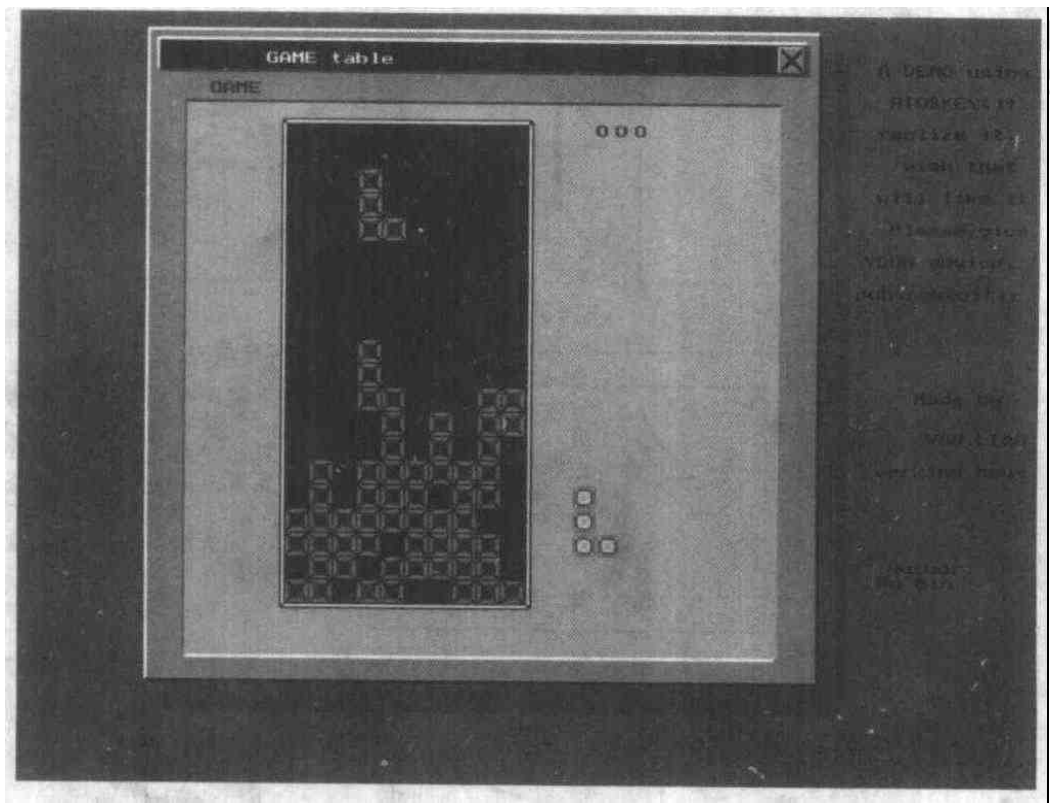


图 16-2　俄罗斯方块游戏（过程中）

16.4.2　视频对战游戏编程思路

视频对战游戏举例：街霸（Street Fighter）。

视频对战游戏的制作重点：

（1）输入设备的及时响应；

（2）同时按键和顺序按键的检测；

（3）游戏对象、动作连贯性和动画效果的创意及精巧设计；

（4）对象间战斗检测以及相应的运算；

（5）对象在游戏中尽量避免闪动。

视频对战游戏的制作难点：

视频对战游戏强调视频效果，保证游戏连贯性尤为重要。这要求在响应输入问题；响应后对象对应动画产生问题；对象间战斗检测问题上都要及时和快速完成。试想，如果一个键盘动作完成后需要延迟很久才会在屏幕上产生响应的动作，这个游戏其实已经失败了。如何在这方面进行提高？这要求我们熟悉硬件情况、代码效率和设计好的算法。比如，键盘响应的及时性中很重要的一点就是省略多余按键（玩视频游戏的人都知道，打到急的时候喜欢拼命按键盘），如果所有按键都去响应键盘就会出于滞后或者等待，屏幕就会出现滞后现象。

视频对战游戏的制作思路要点：

（1）将一半的精力放在画面、对象动画美工方面；

（2）把游戏主循环分为 3 个方面：输入响应、运算检测和对象重绘。分别对每个方面进行提高效率的工作，比如输入响应方面要及时清除键盘缓冲，运算检测的算法精良简单，对象重绘使用后台绘制的双缓冲办法等；

（3）为了增加视频效果，可以将屏幕从前到后分为两到三个层面（在效率允许条件下），分为游戏对象层，游戏背景层，在背景层可以使用调色板动画产生动画效果。

视频对战游戏的制作方案举例（见图 16-3）：

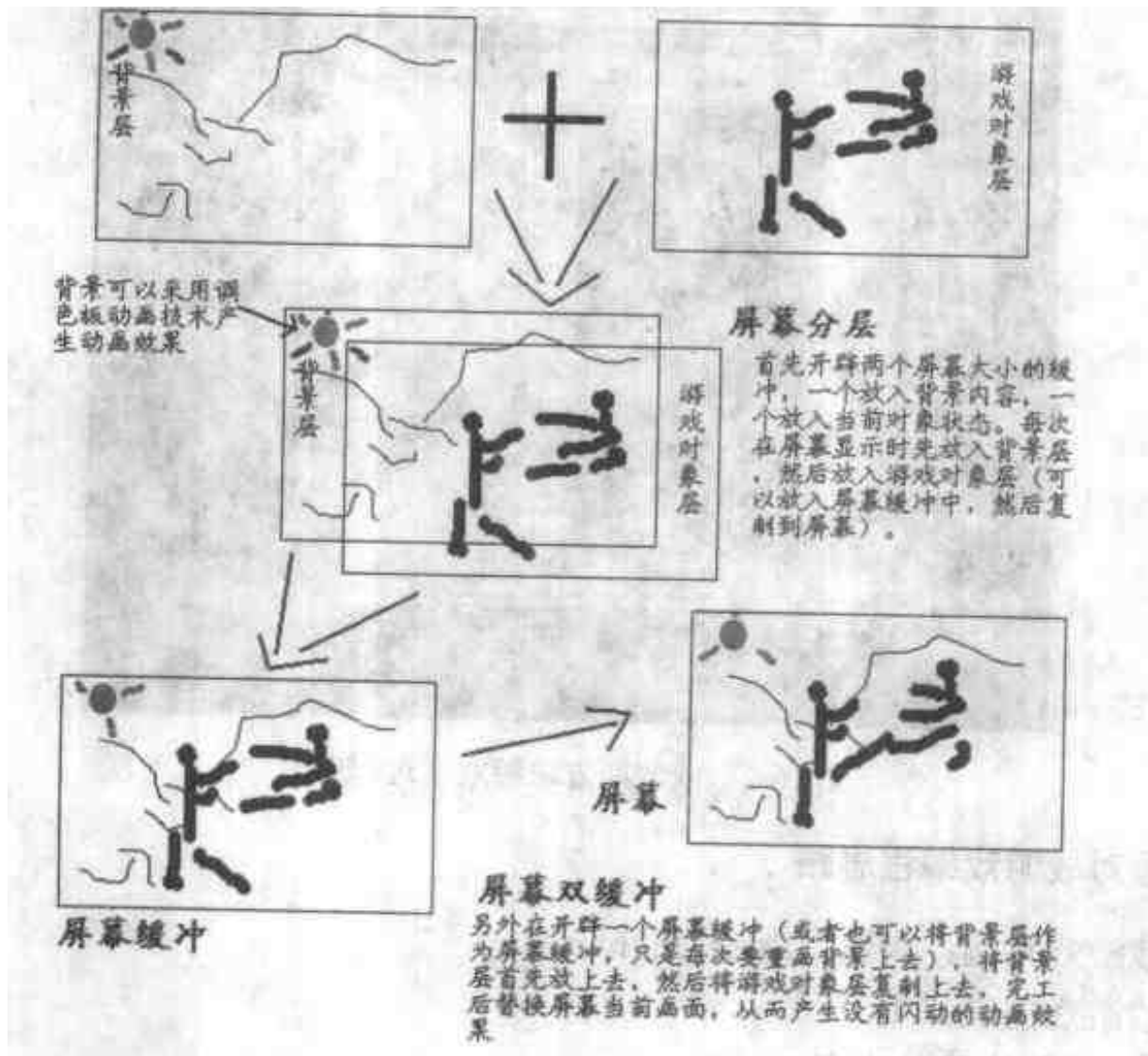


图 16-3　街霸游戏示意图

游戏名：街霸

游戏规则：

（1）两个游戏对象对战，键盘控制对象动作；

（2）允许通过键盘组合键触发复杂动作；

（3）允许产生子弹（比如飞行的火球）类攻击方法。

游戏提高效率办法：

（1）对键盘缓冲进行实时清除和忽略操作；

（2）在子画面技术和屏幕分层技术中（仍然需要子画面，但不需要恢复子画面后的内容）根据实际编程效率比较选择其中一种办法；

（3）在效率允许的条件下再对最终屏幕画面开设双缓冲，避免屏幕出现闪动情况。

16.4.3　魂斗罗类游戏编程思路

此类游戏举例：魂斗罗、阿拉丁。

此类游戏的制作重点：

（1）横向拉屏技术的实现；

（2）屏幕上多子画面的处理，包括游戏对象和场景对象间的交互问题。

此类游戏的制作难点：

魂斗罗类游戏首先通过设计好响应的每关游戏地图，然后通过横向或者纵向拉屏技术来产生游戏对象移动的效果。这类游戏面临的最大难点是如何让游戏对象和当前场景地图中的对象交互起来。比如在魂斗罗中允许游戏对象在多层陆地上和水中移动；游戏对象可以将地图中的桥梁、暗堡和敌人消灭；可以通过捡到子弹标识来获得自己需要的武器装备。必须有精巧的设计思路和良好的数据结构来实现这些功能。

此类游戏的制作流程要点：

（1）横向拉屏技术在游戏循环中起到主线作用；

（2）屏幕中激活的子画面通过链表连接，在每次循环中都根据用户控制和计算机随机控制或计算机预先轨迹设计触发子画面行为而产生新的子画面动画效果；

（3）在每个子画面行为发生后，通过调用碰撞检测函数将其新的行为和其它子画面（包括游戏对象、敌人、子弹等）进行碰撞检测。

此类游戏的制作方案举例：

游戏名：魂斗罗

游戏规则：

（1）铁血战士（游戏对象）要冲过敌人的重重关卡；

（2）游戏对象不能被敌方射击到，否则就死掉了；

（3）游戏对象通过射击敌方将敌方销毁。

实现子画面对象间交互的办法：

（1）（不考虑拉屏问题）在游戏主循环中通过链表调用所有子画面对象行为函数（不采用面向对象思路的事件触发方式），子画面接受键盘输入或者随机控制的触发行为都放在对应子画面的行为函数中进行检测或生成，从而保证所有激活的子画面对象在每次循环中都得到调用；

（2）每个子画面（包括子弹）行为函数在产生行为动作（比如移动、射击等）后，调用针对某一子画面碰撞检测函数，通过对象链表检测此行为动作是否和其他所有子画面产生碰撞，一旦发生碰撞则调用对应的碰撞处理函数（比如销毁、改变子弹等）；

（3）对于游戏对象在地图场景中可以在多层次陆地和水中移动的实现办法，可以将背景地图同时对应于一个二维数组，数组中 1 表示可以支撑，0 表示不可支撑。于是当游戏对象进行上下跳动的时候对游戏对象脚部和地图交叉位置进行数组检测，如果是 1 则游戏对象子画面重画到该位置，如果是 0 则继续检测游戏对象跳动方向下一个交叉位置；

（4）至于地图场景中如此多的子画面如何存放的问题，同样可以通过将背景地图对应于另一个二维数组来实现，数组中通过数字表示子画面类型，进入屏幕的数组对应的子画面即被构建激活。

16.4.4　玛丽、赛车类游戏编程思路

此类游戏举例：玛丽（MARIO）、沙罗曼蛇、坦克大战、淘金者。

此类游戏的制作重点：

（1）此类游戏不使用拉屏技术，而是通过将许多小画面资源组合的方式实现场景的；

（2）需要绘制大量各类单元小画面（比如树、砖头、土地、草地、蓝天等）；

（3）地图实际是一个存放单元小画面序号的二维数组。如图 16-4 所示。

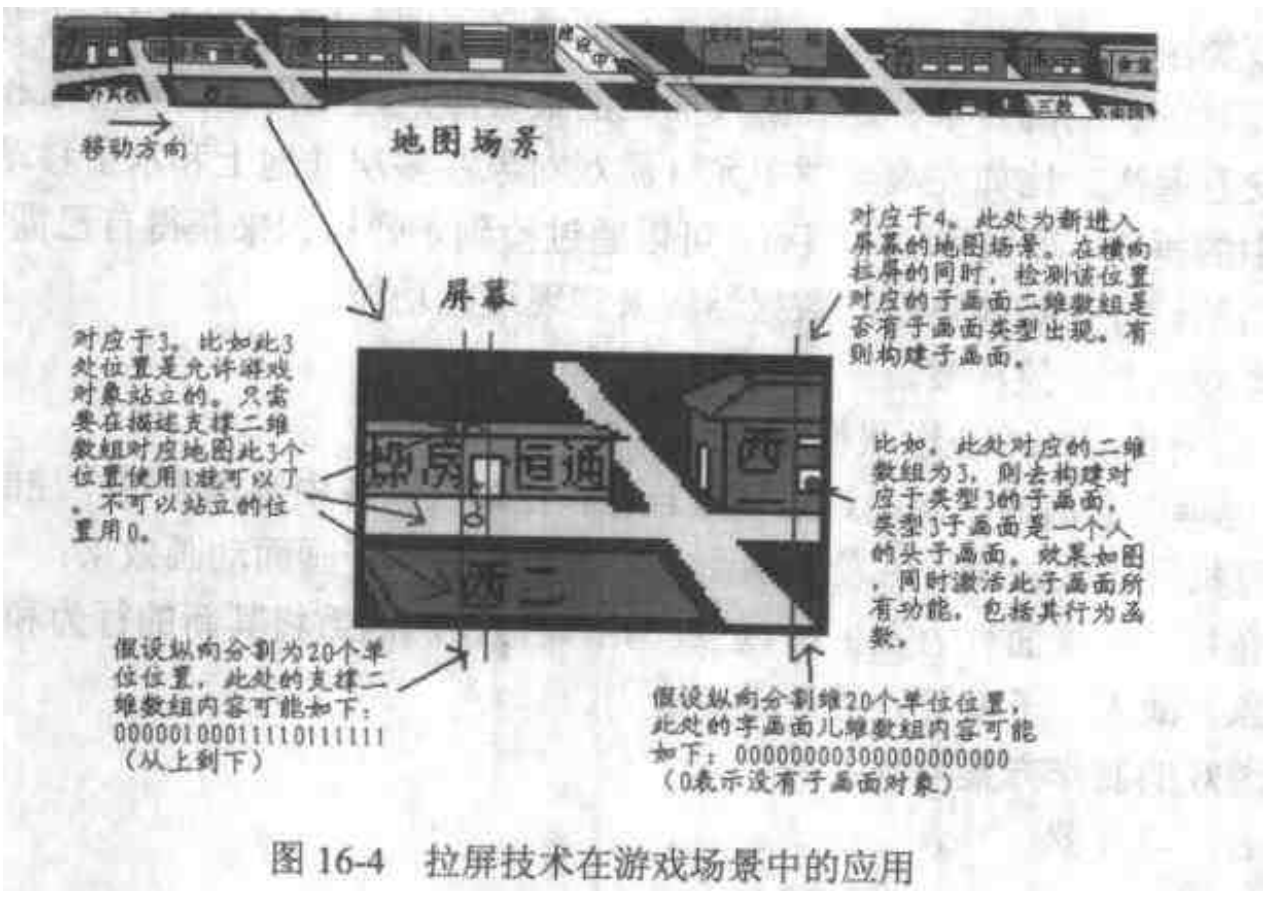


图 16-4　拉屏技术在游戏场景中的应用

此类游戏的制作难点：

此类游戏避免了地图拉屏的问题，但是增加了地图实时构建情况。我们只需要通过构建一个二维地图数组来存放所需要的单元小画面的序号，在新地图进入画面的时候只需要读出二维数组的对应列的内容，然后将序号对应的单元小画面放入屏幕就可以了。此类游戏的制作基本没有什么难度，只是同样要面临魂斗罗游戏面临的对象间的交互问题。

此类游戏的制作思路要点：

（1）通过读取地图数组来实现对屏幕地图的构建；

（2）在游戏主循环中同样采用和魂斗罗类游戏相类似的实现办法。

此类游戏的制作方案举例：

游戏名：玛丽（MARIO）

游戏规则：

（1）玛丽救公主，需要通过一个一个关卡；

（2）玛丽可以通过跳起落下踩扁蘑菇、乌龟等敌人，通过吃金币和特殊能量来实现自我壮大；

（3）一旦玛丽被敌人、敌人的攻击子弹碰到或者落入水中就死亡了。

单位小画面构建场景地图的实现步骤：

（1）绘制各类单位小画面到图像文件，比如可以设定单位像素为 16，于是每个小图片都是 16*16 个象素（如图 16-5）；

（2）依靠单位小画面设计出场景地图，将小画面序号放入地图二维数组对应位置；

（3）将屏幕场景首设置固定背景颜色（比如天蓝色），将单位小画面从文件读取到内存；

（4）根据二维地图数组预先设定序号调用单位小画面到屏幕对应位置（如图 16-6）；

（5）每次地图发生移动的时候，首先将屏幕内场景偏移，然后将新的一列地图数组对应的单位小画面写入场景。

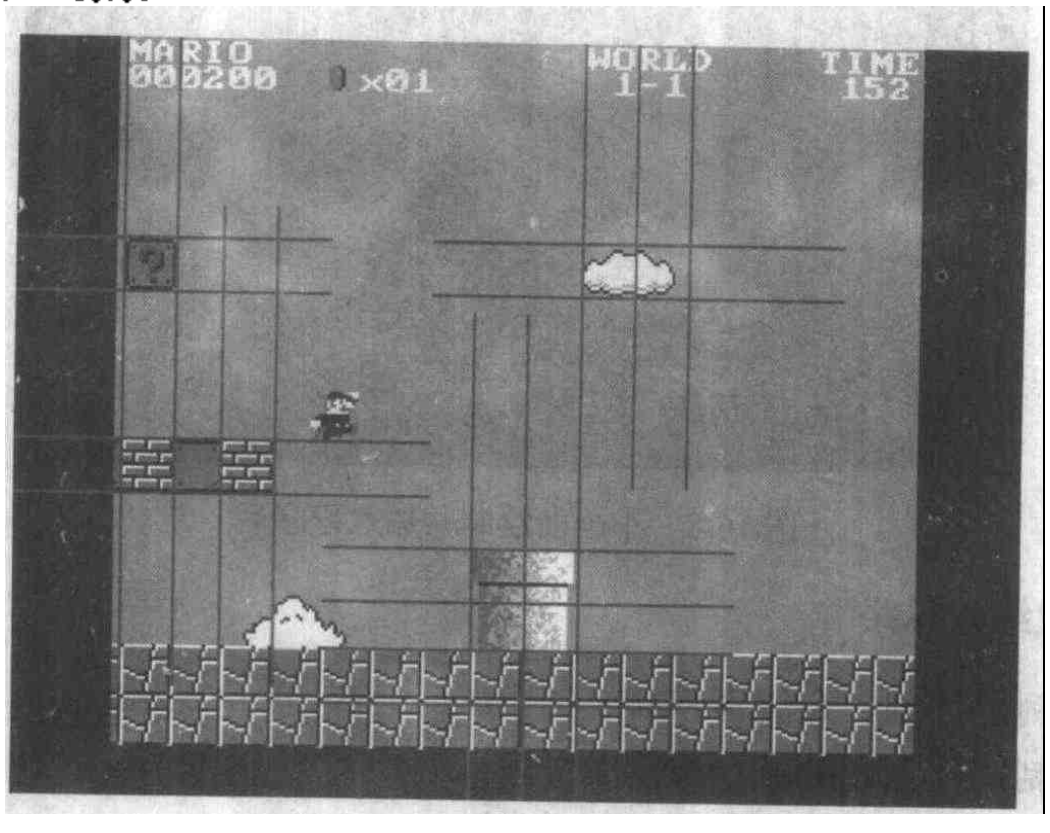


图 16-5　单位小画面

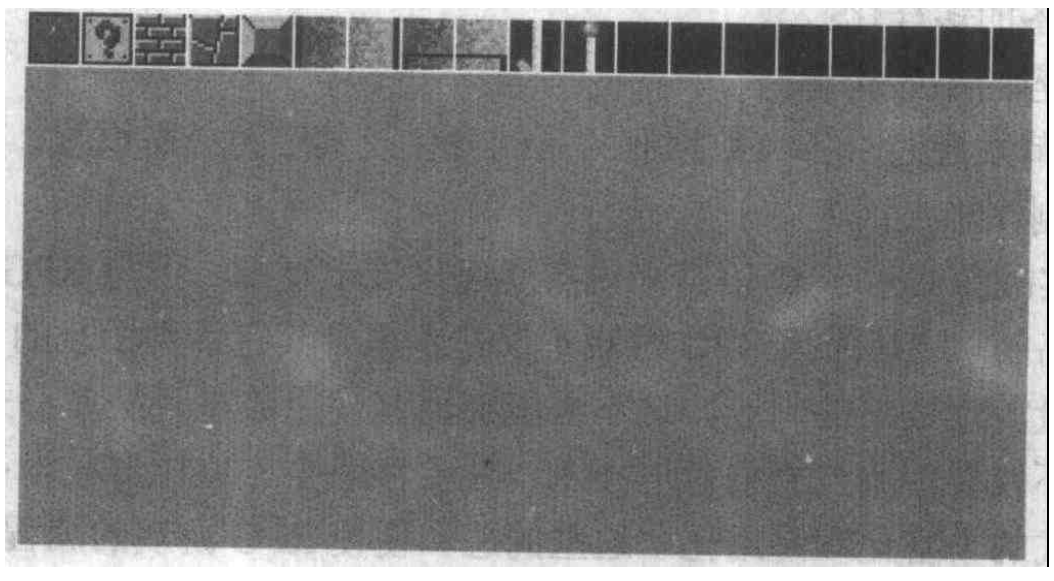


图 16-6　为单位小画面预设序号

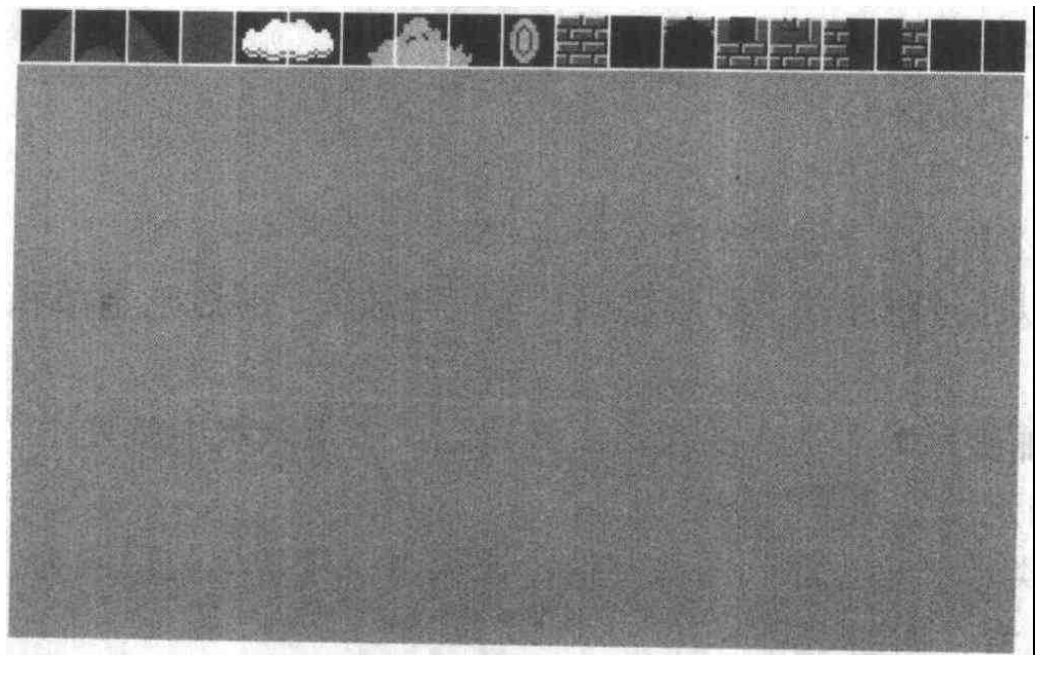


图 16-7　预设序号后



图 16-8　小画面绘制并加入程序后

16.4.5　RPG 游戏编程思路

RPG 游戏举例：仙剑、金庸奇侠传

RPG 游戏的制作重点：

（1）大量数据的存放要求方便调用和修改；

（2）输入设备的多样化；

（3）情节内容的精妙构思；

（4）游戏情节发展时，各类数据间关系的变化及其逻辑关系设置；

RPG 游戏的制作难点：

由于网络是实战游戏对于 C 语言来说实现起来过于复杂，于是 RPG 游戏对 C 语言来说是设计难度较高的。这是因为 RPG 游戏不仅基本集合其它各类游戏的设计难点；同时还增加了要处理大量游戏数据的问题，这不再能够通过使用数组等简单形式来实现了。在拉屏技术方面通常不仅要使用到横向拉屏技术，还要涉及全方位拉屏技术；此外在用户输入方面要求输入方式的丰富化和易用化。这些的总和便使 RPG 游戏的编程制作难上加难。

RPG 游戏的制作思路要点：

（1）虽然在编程实现方面需要花很多功夫，但 RPG 游戏剧本的质量和创新性才是此类游戏能够成功的真正关键所在；

（2）根据剧本要求设计出良好的数据结构和编程实现方案，在此基础上建立一整套适合游戏的数据库，数据库中还应该包括数据之间的逻辑关系和数据在游戏场景地图中的出现位置等问题；

（3）在游戏中使用对象事件触发方式非常有利于游戏的修改和扩充，同时在大量代码的情况下减少了出错的可能和增加了程序的可读性。

16.5　本 章 小 结

TSR 程序就是终止并常驻内存程序（Terminate and Stay Residemt）的简称。要实现驻留程序首先必须了解中断改写的方法：

（1）保存所要改写的中断向量入口；

（2）将中断向量入口指向新的中断服务函数；

（3）在程序结束后恢复中断向量入口。

驻留程序通常的制作过程：

（1）改写中断函数；

（2）使用 keep()函数将当前程序驻留在内存空间中。

文件感染型病毒主要的两件事情：

（1）让程序能够将自己在不影响其它程序工作的情况下复制给它们，使它具备继续复制的能力；

（2）在一定条件下使其产生某种发作效果。

对于 C 语言来说它的实现过程：

（1）主程序调用病毒功能函数；

（2）病毒功能函数读取查找同目录下所有 C 文件；

（3）找到一个（被感染 C 文件），打开它，并且将此文件全部读取到数组变量；

（4）重新创建一个同名文件（被感染 C 文件）；

（5）数组变量写回此被感染 C 文件，在这过程中将病毒源文件所需要的头文件、病毒功能函数调用语句同时写入；

（6）打开病毒源文件，将病毒功能函数全部写到被感染 C 文件的最后。

面向对象思路在程序中主要表现为：

（1）在描述对象的结构体中增加了事件函数指针和对象链表；

（2）在游戏前进行对象构建；

（3）游戏主循环通过检测对象事件触发条件（通过鼠标和键盘）来进行，一旦有事件被触发就去调用该对象的事件函数；

（4）对于没有用的对象进行释放。

从 C 语言制作游戏的角度，游戏类型主要分为：

（1）桌面游戏；

（2）视频战斗游戏；

（3）魂斗罗类游戏；

（4）玛丽、赛车类游戏；

（5）RPG 游戏。

各类游戏都有各自的制作思路、游戏规则、难点和重点，我们必须分类考虑其解决与实现的办法。

学后建议

（1）考虑游戏中还有哪些比较重要的内容没有被考虑到，还有哪些东西你比较感兴趣，然后拿出来研究（比如多任务的实现、加密技术的应用和三维游戏制作）；

（2）尝试用 C 语言制作无害的警告类型病毒，然后加入你的游戏软件中，一旦别人对你的游戏非法盗版或者试用过期了就跳出警告；

（3）尝试用本书学到的知识制作各类型的游戏，然后在实际制作过程中扩充。

第 17 章　游 戏 例 程

本章导读

前面的章节对于游戏编程的介绍都是从某个具体角度切入的，在大致掌握了这些知识以后现在可以将它们整合起来尝试制作较为完整的游戏了。本章将介绍一个简单 RPG 游戏的制作过程。

在上一章 4.5 节介绍了 RPG 游戏的编程思路。事实上 RPG 游戏将涉及到接口技术、中文显示技术、拉屏技术、子画面动画技术和数据库技术等一系列游戏编程技术。

为了将本书制作的所有游戏函数充分发挥作用，本章制作了一个游戏函数库，将各类游戏函数分列头文件中，然后由一个总的头文件进行声明。

本章重点

（1）将本书各章中制作的游戏函数编制成游戏函数库，对库中所有游戏函数的掌握都可以按照你的游戏的要求进行修改；

（2）学习相对较复杂的游戏制作流程，掌握制作中游戏规则设计、数据文件、数据文件的制作办法和程序流程设计。

17.1　建立通用游戏函数库

制作这个游戏将使用到前面各章节的技术和函数。事实上，在以后制作所有游戏的时候都要用到这些游戏函数，所以制作一个通用的游戏函数库是非常有必要的。为了增加这个函数库的可读性我考虑将每个章节中差不多类型的函数放在一个头文件中，然后使用一个头文件 all.h 对所有游戏头文件进行声明，在游戏中只需要调用 all.h 头文件就可以使用函数库了。

我在附录 E 中注明的 200 多个游戏函数分类放入 36 个头文件，然后使用 all.h（请查阅所附光盘的“source\17\h”目录）对这些头文件和标准头文件以及一些全局变量和常量进行声明和定义。

在具体的游戏中，我们通常不会使用到其中所有的游戏头文件，于是可以通过改写 all.h 的方式来注释掉那些不会被使用的头文件从而提高游戏的编译质量和运行效率。

17.2　游 戏 创 意

由于本章考虑通过设计一个准 RPG 游戏来贯穿本书之前的章节，所以在游戏创意方面更多考虑的并不是游戏情节的新颖而是寄希望于使用到足够多的游戏编程技术。

1. 游戏情节

某大学男生（男主人公“猪头”）在学校网络中心上网聊天的时候，认识了一个同校的女生（女主人公“红色头发”）。之后女主人公决定从网络中消失，男主人公就开始在学校

（续表）

信息采集地	获取方式	信息内容	信息价值
西部一号教室	和“樱木”同学交谈	你就是我要找的红色头发;兄弟你同性恋呀;阿;认错人了;我倒	无用
	和“小红”同学交谈	你叫红色头发吧;是;我是猪头;不认识;昨天还一起聊天;可能别人用了我的机器;麻烦告诉我机器号;六号	知道红色头发并非真正的女主人公，找到下一个前往方向：网络中心六号机器
	和“残云”同学交谈	你的网名是不是红色头发;拜托你是色盲阿;哦;看错了	无用
	和“阿甘”同学交谈	阿黄借我你的上网卡;里面只有十元了;我会加钱的;取到上网卡一张	获得去网络中心上网时使用的上网卡
	和“木村”同学交谈	帅哥我们寝室钥匙你带了么;当然带了;西门在睡觉我等会回去自己开门;给我;取到寝室钥匙	获得寝室钥匙，为之后去西部八号寝室楼铺垫
网络中心	旁白	老兄有一点创意好不好;说六号就点就当真是六号啦;下次放病毒程序在这里;好秀逗;我本来就是猪头嘛	无用
	六号机器记录	昨天三人使用;计算机的西门咳嗽;数科的六月冰;人文的西子月;前两个我认识;先到西八问咳嗽吧	获得两个调查对象：西门咳嗽和六月冰，找到下一个前往场景：西部八号寝室楼，同时知道女主角是“西子月”
西部八号寝室楼	和“咳嗽”同学交谈	起来老师来检查了;把我从窗子扔出去;昨天你上网的机器后来谁用的;六月;之后呢;打电话问六月吧	获得打电话向六月冰询问的暗示
	通过电话听筒和“六月冰”寝室同学交谈	摩西摩西;喂;请问六月冰在么;去操场体锻敲卡了;什么时候回来;刚走;谢;再见	找到下一个前往场景：大操场
	照镜子	;魔镜魔镜告诉我西子到底是什么;俗;没什么对不起你呀;整天照照照都给你累死了;显示器都没说过我呢	无用
	桌上的体锻“活动卡”	正好这里有一张活动卡;顺便去敲章;得到活动卡	或者进入大操场凭证的活动卡
大操场	和“六月冰”同学交谈	怎么只你一个;我慢呀;昨天上好网谁坐你位置的;不认识的;她是中文的我找她;我陪你去文苑楼找	决定去下一个场景：文苑楼

（续表）

信息采集地	获取方式	信息内容	信息价值
文苑楼	和“六月冰”同学交谈	大师兄好像他们都下课了;看来找不到了;不会的努力呀;阿;我知道了;小六你先回去休息吧	场景中男主人公在大楼上面对着学思湖，联想“西子月”的名字，决定去下一个场景：学思湖
学思湖	地上的未知物	脚下是什么呀;原来是 一张电影票;反面还有字;美酒春花西子月相携却是梦中人;真的没猜错	确认西子月在这里留下了一张电影票
东部大礼堂	看电影播放内容	痞子我走了;我明白了;游戏结束了	最终知道女主人公离开网络的原因是网络过于虚幻，无法寄托

4. 游戏使用到的技术

考虑到目前计算机操作系统繁多，DOS 下程序常常在不同 Windows 平台下出现不同问题。于是决定避免使用容易产生问题的声音和背景音乐技术、扩展和扩充内存技术。见表 17-2。

表 17-2　采用不同平台下的兼容技术

技术名称	解决问题
中文显示技术	显示对白内容
BMP、JPG 图形文件调用	显示地图和场景等
调色板动画技术	闪烁星星的实现和淡入、淡出实现
横向拉屏技术	学校地图的实现
子画面技术	地图内主人公行走
进度文件保存和调用技术	保存和调用游戏进度
数据库类型文件调用技术	游戏场景、对象、对象行为的调用
键盘缓冲清除技术	避免主人公控制之后
图像鼠标技术	用主人公形象作为鼠标外形
界面技术	便于游戏操作

17.3　游 戏 规 划

17.3.1　详细设计

1. 游戏情节发展流程（见图 17-1）

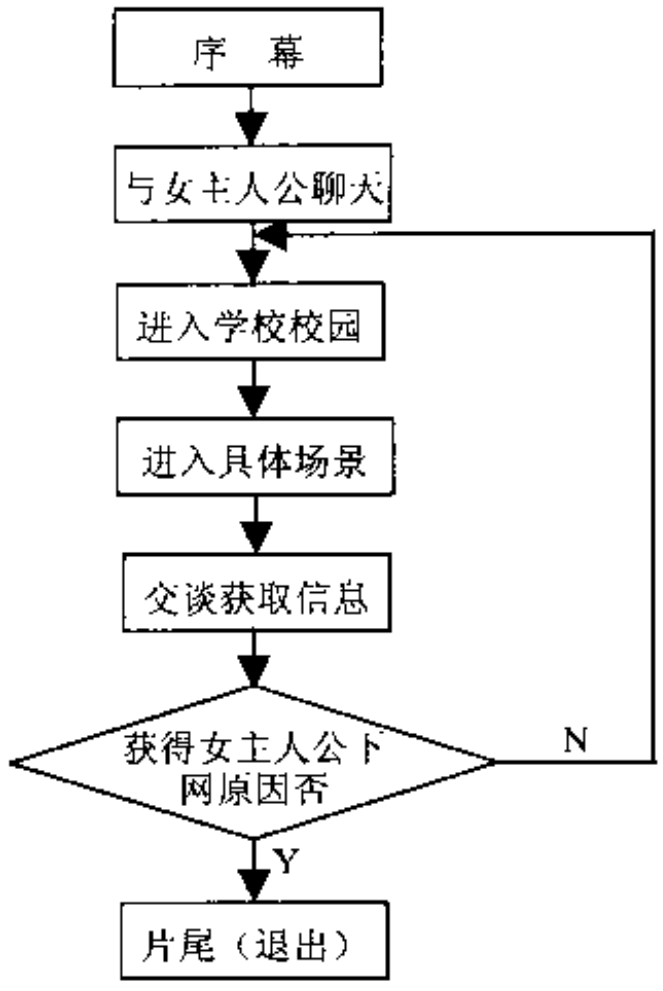


图 17-1　游戏流程示意

2. 用户操作规则（见表 17-3）

表 17-3　游戏操作规则

任务	操作规则
进入下一页面	键盘任意键
大地图内男主人公行走	键盘 a(向左)d(向右)
进入具体场景	键盘 w(向上)x(向下)
退出游戏	键盘 Q 键
大地图内菜单触发和按钮触发	鼠标移动加鼠标左键
具体场景内男主人公移动	鼠标移动
与场景内对象交谈获取信息	鼠标左键（热点在男主人公的点金杖、手或者脚）
看下一句交谈内容	键盘任意键
退出具体场景回到大地图	鼠标右键

3. 图形鼠标热点

由于在场景内人物是由图形鼠标控制的，而图形鼠标大小为 65*50 非常的大。于是在具体场景内必须设置多个热点才能够让其和场景内任意位置对象发生交谈。这里我们设置男主人公的点金杖、右手或者右脚作为热点，在不同的时候通过不同的热点和对象发生接触，如图 17-2 所示。



图 17-2　人物“猪头”

4. 获得物品的途径

表 17-4　获取物品途径

获得物品场景	获得物品的途径	获得物品
东部食堂	和“大师兄”同学交谈	10 元饭菜票
西部·一号教室	和“木村”同学交谈	寝室钥匙
西部一号教室	和“阿甘”同学交谈	上网卡
西部八号寝室楼	在桌子上面	活动卡
学思湖畔	在亭子边的地上	电影票

5. 情节发展条件列表

表 17-5　情节发展条件

情节发展内容	需要物品
进入网络中心	上网卡
进入西部八号寝室楼	钥匙
进入东部大礼堂	电影票
进入大操场	体锻活动卡
进入东二寝室楼（隐藏版）	钱

6. 场景设计

为了使本游戏在更多机器上能够运行，决定避免使用大内存（扩充、扩展内存技术）调用技术。于是要求能够直接申请游戏地图大小的常规内存。我制作的游戏地图的大小为 640*50，还不到段内最大可申请内存（64k）的一半，一定可以申请到，见图 17-3。



图 17-3　游戏地图

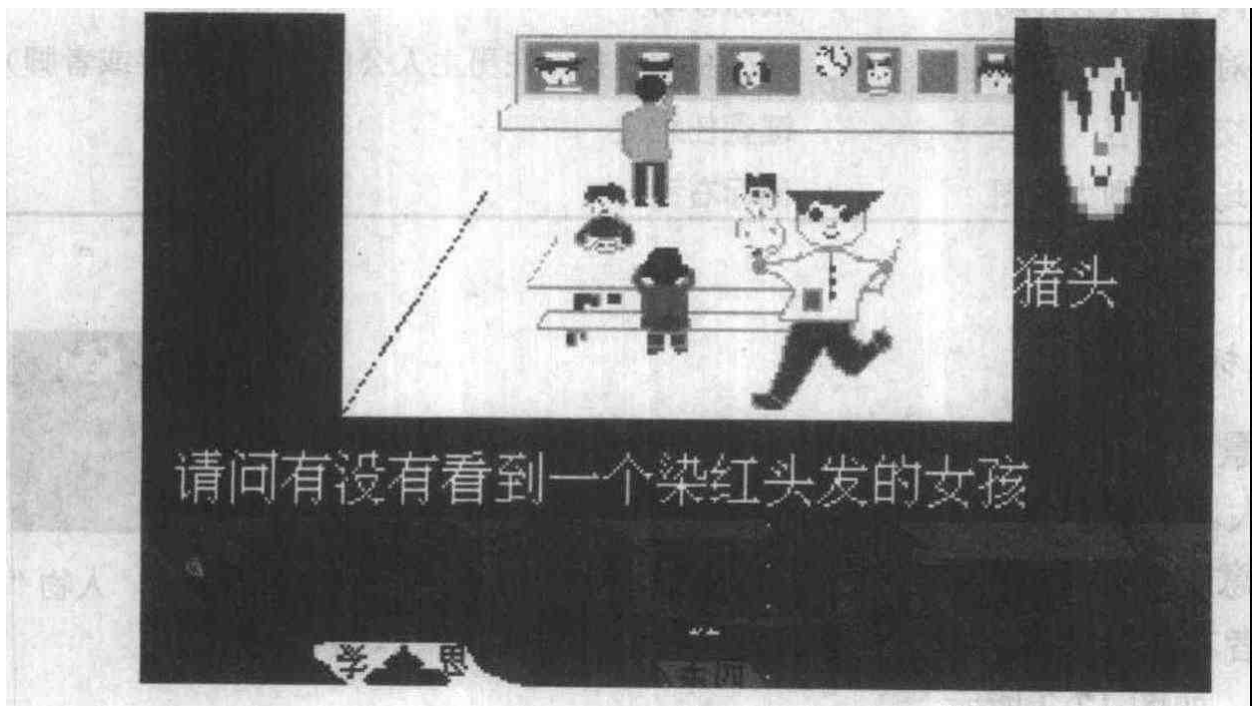


图 17-4　游戏场景

同时由于我们采用的视频模式 13 屏幕宽度只有 320，而地图宽度为 640。于是地图在屏幕中只能够显示一半，必须通过男主人公在地图上的移动（横向拉屏技术）使得所有地图在屏幕中显示。屏幕高度为 200，而地图的高度只有 50，于是决定将地图放在屏幕的最下端显示，其上空余的 320*150 区域用于显示具体场景和人物对话。如图 17-4 所示。

其中从坐标（50,0）到（269，119）的范围用于显示具体场景画面；（0,0）到(49,59)和(270,0)到(319,59)的范围用于显示两个对话人物头像，在头像下方显示人物名称；(0,120)到（319，149）的范围内显示谈话内容；最下面的地图则正好是从(0,150)到(319,199)。

17.3.2　程序流程设计

在着手编写游戏代码前，首先必须根据游戏策划和详细设计来编写出游戏流程。事实上如此大的游戏已经很难简单地使用流程语言中的流程设计方法了。从第 10 章子画面技术到第 15 章界面技术我们在程序中不断加强面向对象思路，在较大的游戏编程中融入面向对象、事件触发的思路可以大大提高程序设计的效率。事实上，我深切地感受到真正热爱而坚持深入 C 语言编写的人最终也一定会自己走到面向对象这一步的，我猜想 C++本来就是源自于 C 语言最初探索者在编写大程序中的痛苦和顿悟。我是一个深爱 C 语言的人，即使是在它已经被基本淘汰以后，可是写到这一章我才终于决定转向 C++。

不过，我们还必须以流程作为主线，将对象事件触发加入其中。事实上，大部分对象和事件触发甚至还无法成型。我写 C 语言游戏的感受是，在复杂的地方考虑构建对象和用事件触发，而在简单的地方仍然可以使用流程性的思路加以解决。

程序流程(从面向对象、事件触发角度)见图 17-5 所示。

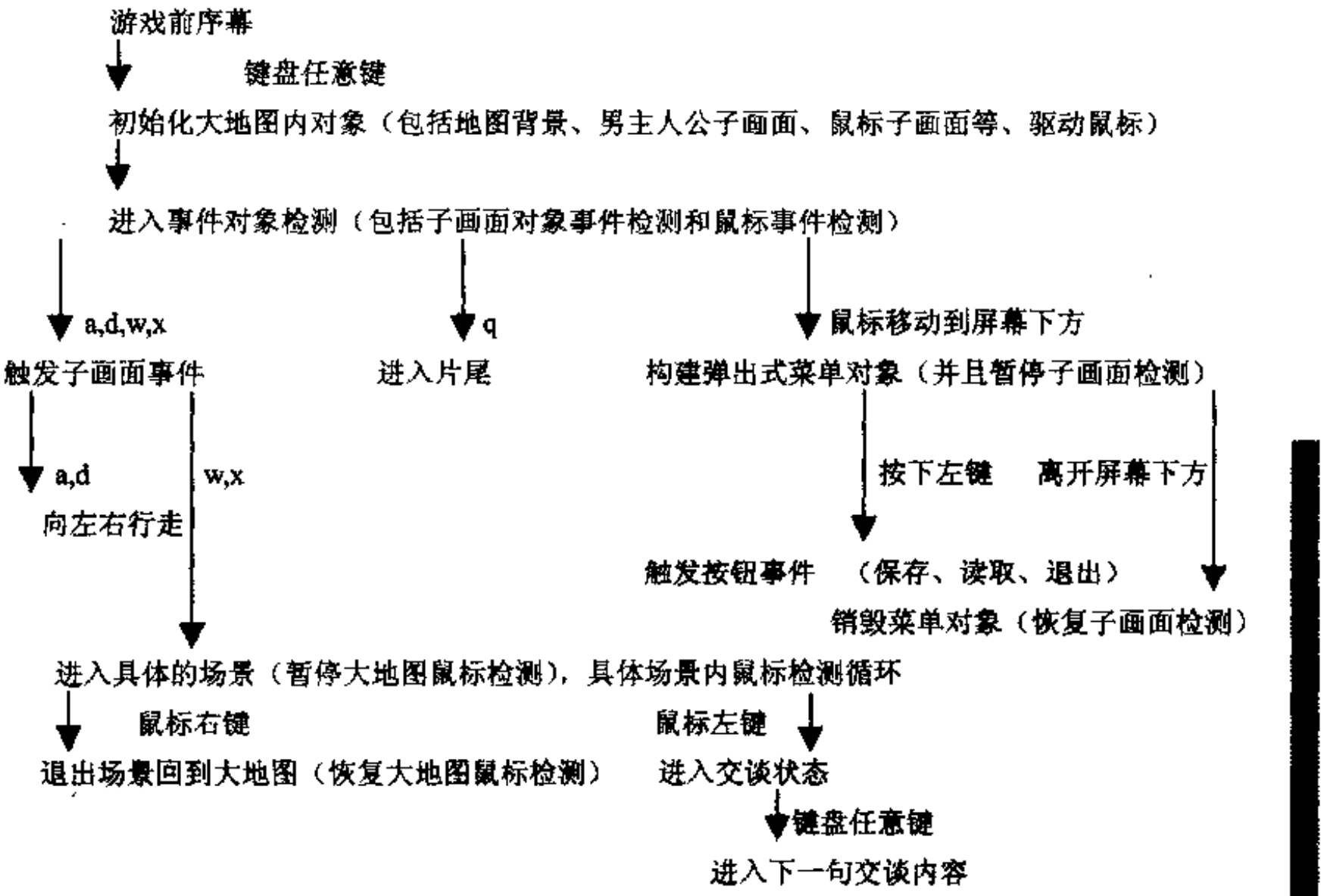


图 17-5　程序流程示意

17.4　程 序 编 写

17.4.1　文件清单

本游戏一共包括 47 个文件和一个游戏函数库（库中有 37 个头文件）。本游戏中除了*.c 和*.h 代码文件，还包括数据文件和进度文件，以及许多图像文件。表 17-6 是游戏文件清单。

表 17-6　游戏文件

文件名称	介绍
Rpg.exe	可执行游戏程序
Rpg.c	游戏主程序
All.h	游戏函数库调用头文件
Rpg.h	部分改写的 rpg.c 专用的游戏函数
Load.dat	游戏进度文件
Room.dat	场景位置、调用图片、进入条件、对话位置和获得物品数据清单
Chat.dat	对话对象和对话内容清单
Hzk16	中文点阵字库文件
*.bmp	序幕图像
*.jpg	地图、游戏具体场景、人物对象、鼠标对象图像

17.4.2　进度文件

RPG 游戏通常周期比较长，玩者需要花费很多时间才能够将整个游戏进行完。

在第 11 章介绍了进度文件的保存和读取办法，本游戏中只需要当触发菜单对应按钮的时候调用进度读取和进度保存函数就可以了。本游戏中的进度变量数据很少，除了男主人公在大地图中的坐标位置就只有 5 个物品是否被取到，一共 6 个数据需要保存。对于前者我们只需要将程序中表示大地图在屏幕内位置的变量（sx）对应起来就可以了，而后 5 个物品则需要用一个数组变量（my_things[]）和其对应，同时 0 表示该物品还没有被取得，1 表示取得物品。以下数据是从本游戏进度文件(loatd.dat)取出的数据：

342;0;1;0;0;0;

“342”表示当前主人公在地图中的位置是 342（地图范围是从 0~640）。“0;1;0;0;0;”则分别对应"money"（饭菜票）、"ticket"（电影票）、"card"（上网卡）、"key"（寝室钥匙）和"sport"（体锻卡），也就是说目前男主人公只拥有一张电影票，而其他 4 样物品都没有取得。

17.4.3　图片文件

游戏中使用到最多的就是图片文件，一共有 39 个，其中 bmp 文件有 2 个，其它的都是 pcx 文件，所有的图片文件都是 256 色的。Bmp 文件用于显示一个实地拍摄的校园风景制作的游戏开场画面和一个网络聊天室背景画面。Pcx 文件被用于 9 个场景画面和 28 个人物头像以及物品画面。

这些图片的制作用去了我相当多的时间，不过除了工作量较大以外制作难度是非常低

当然如果你真正要制作有人喜欢玩的游戏，就需要专业的美工技术。我这里只是使用画图软件，运用第 10 章画面绘制中的简单方法实现的。

17.4.4　数据文件

在数据文件设计的问题上，我花费了不少脑筋。由于没有使用 dbf 数据库文件存放数据，于是必须依照第 11 章数据文件设计的方法自己解决数据存放规则和读取的问题。

本游戏中主要存放的固有数据内容包括两大块。一块是与具体场景相关的数据，一块是与谈话相关的数据。当男主人公在大地图要求进入场景的时候调用场景相关数据；而当男主人公在具体场景内要求和别人对话的时候先调用场景相关数据中的对话位置数据，如果有这个对话位置存在再去调用对话相关数据。为了方便其间将这两块数据分别存放于两个数据文件 room.dat 和 chat.dat。

场景相关的数据包括：

（1）场景在大地图内起始位置；

（2）场景在大地图内结束位置（相对起始位置的相对位置）；

（3）场景在大地图上方或者下方（上方用 0 表示，下方用 1 表示）；

（4）进入该场景需要的物品名称（如果不需要物品则用 null 表示）；

（5）调用该场景的图片名称；

（6）该场景内对话对象数量；

（7）该场景内对话位置（有多少对话对象就有多少对话位置）：

a. 起始位置 x 绝对坐标*2(乘以 2 的原因是鼠标在其中横坐标移动单位是像素的 2 倍)；

b. 起始位置 y 绝对坐标；

c. 结束位置 x 绝对坐标*2；

e. 结束位置 y 绝对坐标；

f. 此位置对话是否能够获得物品（如果不能获得物品则用 null 表示）。

按照第 11 章数据文件设计方案，我们使用“分号”作为数据间分割符号，而用“等号”作为数据段间的分割符号。同时对于场景数据，我们决定在给每个场景数据段分配 192 个字节的空间，也就是说第 3 个场景的数据段是从文件内 2*192 偏移开始的。场景数据 room.dat 请查阅所附光盘的“source\17\game”目录。

以第三个场景为例：

```
470;50;0;key;west8.pcx;4;100;5;160;35;null;205;15;230;35;null;405;10;450;40;null;330;
50;355;55;sport;=
```

（1）场景在大地图内起始位置（470）；

（2）场景在大地图内结束位置（相对起始位置的相对位置）为 50；

（3）场景在大地图上方或者下方（0 为下方）；

（4）进入该场景需要的物品名称（key 为寝室钥匙）；

（5）调用该场景的图片名称 west8.pcx（如图 17-6 所示）；

（6）该场景内对话对象数量为 4；

（7）该场景内对话位置：

1*起始位置 x 绝对坐标*2：100；
1*起始位置 y 绝对坐标：5；
1*结束位置 x 绝对坐标*2：160；
1*结束位置 y 绝对坐标：35；
1*此位置对话是否能够获得物品：null；
2*起始位置 x 绝对坐标*2：205；
2*起始位置 y 绝对坐标：15；
2*结束位置 x 绝对坐标*2：230；
2*结束位置 y 绝对坐标：35；
2*此位置对话是否能够获得物品：null；
3*起始位置 x 绝对坐标*2：405；
3*起始位置 y 绝对坐标：10；
3*结束位置 x 绝对坐标*2：450；
3*结束位置 y 绝对坐标：40；
3*此位置对话是否能够获得物品：null；
4*起始位置 x 绝对坐标*2：330；
4*起始位置 y 绝对坐标：50；
4*结束位置 x 绝对坐标*2：355；
4*结束位置 y 绝对坐标：55；
4*此位置对话是否能够获得物品,sport；

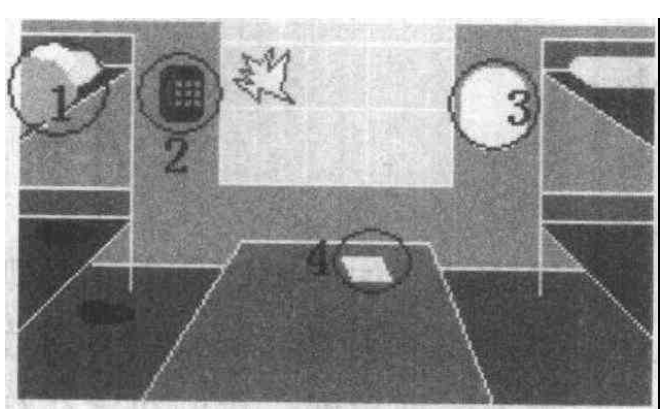


图 17-6　场景图片 west8.pcx

在这个场景中一共和四个对象发生对话。根据对话数据显示，第一个对话对象是在睡觉的西门咳嗽，第二个是电话机，第三个是镜子；第四个是桌上的体锻活动卡。同时，在与第四个对象体锻活动卡接触后，男主人公身上就增加了体锻活动卡这个物品。

谈话数据包括场景内谈话对象（有多少谈话位置就有多少谈话对象）：

（1）第一个谈话对象对应的图形文件名称；
（2）第二个谈话对象对应的图形文件名称；
（3）第一个谈话对象的名称；
（4）第一个谈话对象的名称；
（5）对话语句数；
（6）详细对话（有多少对话语句就有多少详细对话）：

a. 说话人代号（1 表示弟一个谈话对象，2 表示第二个谈话对象）；

b. 说话内容；

值得一提的是，谈话数据的文件结构设计和房间数据基本相同，只是给每个场景对话数据段的空间预留为 1152 字节，同时给每个场景中每个对象对话预留了 192 字节的空间。在所有数据间使用“分号”分割；在每个场景对话数据段结束位置使用“等号”；同时也在场景内每个对象对话结束位置使用“等号”。比如，前面西八寝室对话，只需要根据 room.dat 知道此场景是第三个场景，然后到 chat.dat 中寻址 2*1152 开始就是西八寝室对话开始位置了；而西八寝室对话发生了 4 次，于是，如果是第 3 次和镜子对话则再在 2*1152 基础上遍历 2*192 就可以到达和镜子对话的起始位置了。谈话数据 chat.dat 请查阅所附光盘的“source\17\game”目录。以第三数据段对话为例：

head.pcx;head8.pcx;猪头;咳嗽;6;1;起来老师来检查了;2;把我从窗子扔出去;1;昨天你上网的机器后来谁用的;2;六月;1;之后呢;2;打电话问六月吧;=

head.pcx;head9.pcx;猪头;听筒;8;2;摩西摩西;1;喂;1;请问六月冰在么;2;去操场体锻敲卡了;1;什么时候回来;2;刚走;1;谢;2;再见;=

head.pcx;head10.pcx;猪头;镜子;5;1;魔镜魔镜告诉我西子到底是什么;2;俗;1;没什么对不起你呀;2;整天照照照都给你累死了;1;显示器都没说过我呢;=

head.pcx;head17.pcx;猪头;活动卡;3;1;正好这里有一张活动卡;1;顺便去敲章;1;得到活动卡;;= ;=

取其中第三个对象对话内容：

head.pcx;head10.pcx;猪头;镜子;5;1;魔镜魔镜告诉我西子到底是什么;2;俗;1;没什么对不起你呀;2;整天照照照都给你累死了;1;显示器都没说过我呢;-

（1）第一个谈话对象对应的图形文件 head.pcx；
（2）第二个谈话对象对应的图形文件 head10.pcx；
（3）第一个谈话对象的名称为猪头；
（4）第一个谈话对象的名称为镜子；
（5）对话语句数为 5；
（6）详细对话：

1*说话人代号：1；
1*说话内容：魔镜魔镜告诉我西子到底是什么；
2*说话人代号：2；
2*说话内容：俗；
3*说话人代号：1；
3*说话内容：没什么对不起你呀；
4*说话人代号：2；
4*说话内容：整天照照照都给你累死了；
5*说话人代号：1；
5*说话内容：显示器都没说过我呢。

对应的 head.pcx 和 head10.pcx 如图 17-7 所示。

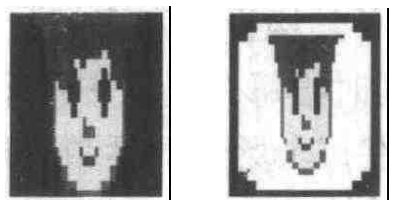


图 17-7　头像图标

17.4.5　代码文件

对于代码文件我将只做注释而不进行详细讲解，因为在之前的游戏设计过程中已经从各方面表述的比较明确了，同时代码函数模块也写的比较清楚。以下是我在代码编写时的一些实际情况：

（1）代码编写是完全根据程序流程设计而进行顺序编写的；

（2）代码编写是基于游戏函数库进行函数模块编写的；

（3）改写 all.h 游戏函数库，注释掉了一些不使用的函数；

（4）增加了 rpg.h 函数，用以改写或新写单独服务于本游戏的函数；

（5）在代码编写过程中不断修改、优化数据文件的存储方式；

（6）场景和谈话的数据文件读取是最繁琐的编程内容；

（7）从游戏创意到图画设计到游戏编写全程历时 2 周左右，后修改历时 1 周左右。

主要函数：

```
void Pre_Game(void); //游戏前
void Game_Load(void); //游戏中
void Game_End(void); //游戏后
void Build_Screen(void); //初始化大地图场景
void Judge_Sprite(void); //大地图中男主人公事件检测
void Judge_Object_RPG(void); //大地图中鼠标事件检测
void Boy(sprite_ptr sprite); //男主人公对象事件函数
void Room(int direction); //具体场景调用函数
void Chat(int x,int y,int seek); //谈话触发函数
void hel(windows_ptr win); //帮助按钮事件
void loa(windows_ptr win); //读取游戏进度按钮事件
void sav(windows_ptr win); //保存游戏进度按钮事件
void men_RPG(windows_ptr win); //弹出菜单事件
void Build_Object_RPG(); //构建游戏任务栏界面
```

主程序 rpg.c 以及头文件 rpg.h 和 all.h 请查阅所附光盘的“source\17\game”目录或“h”目录。

17.5　游 戏 场 景

游戏的各个场景画面如图 17-8~图 17-19 所示。



图 17-8　封面

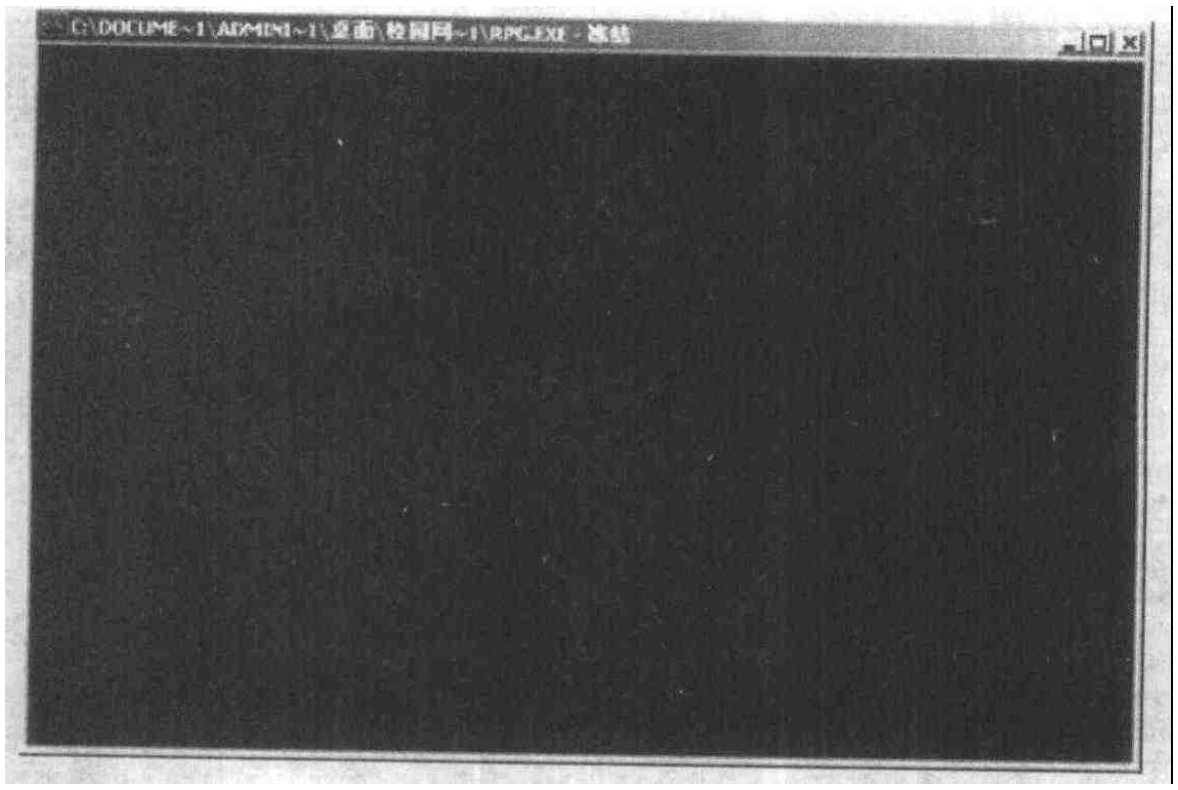


图 17-9　星星 1

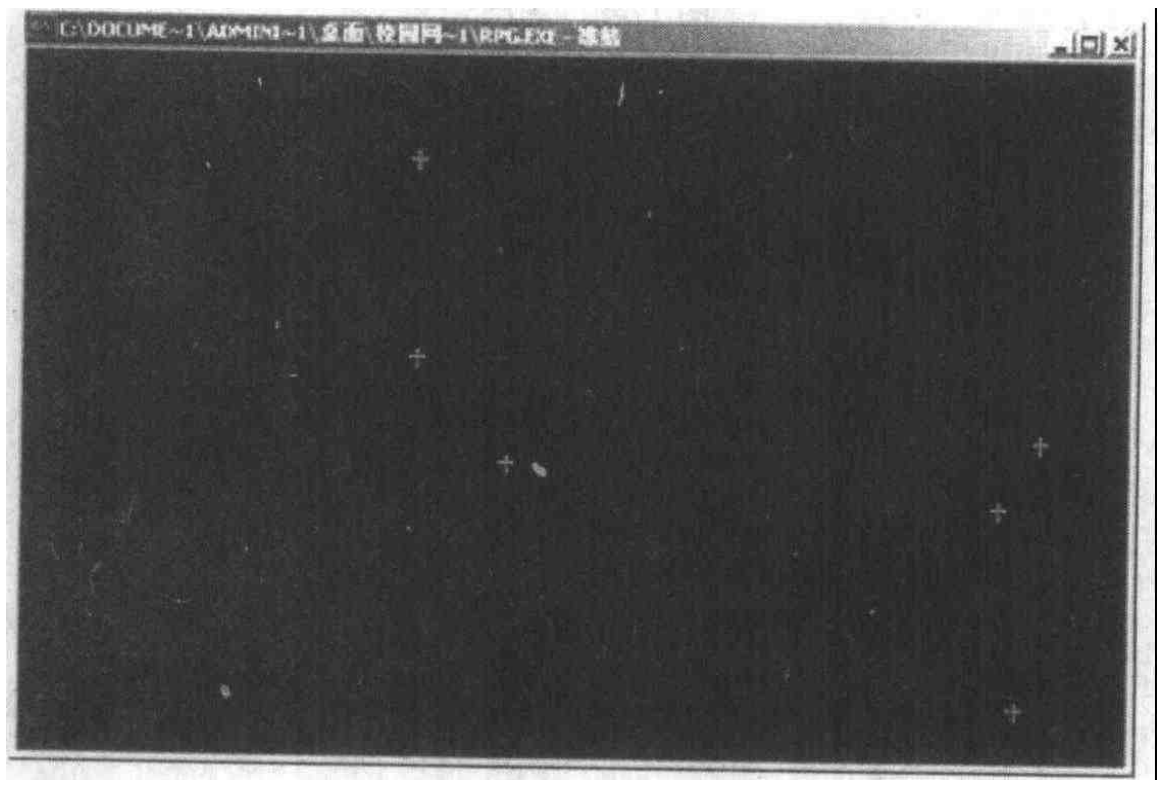


图 17-10　星星 2

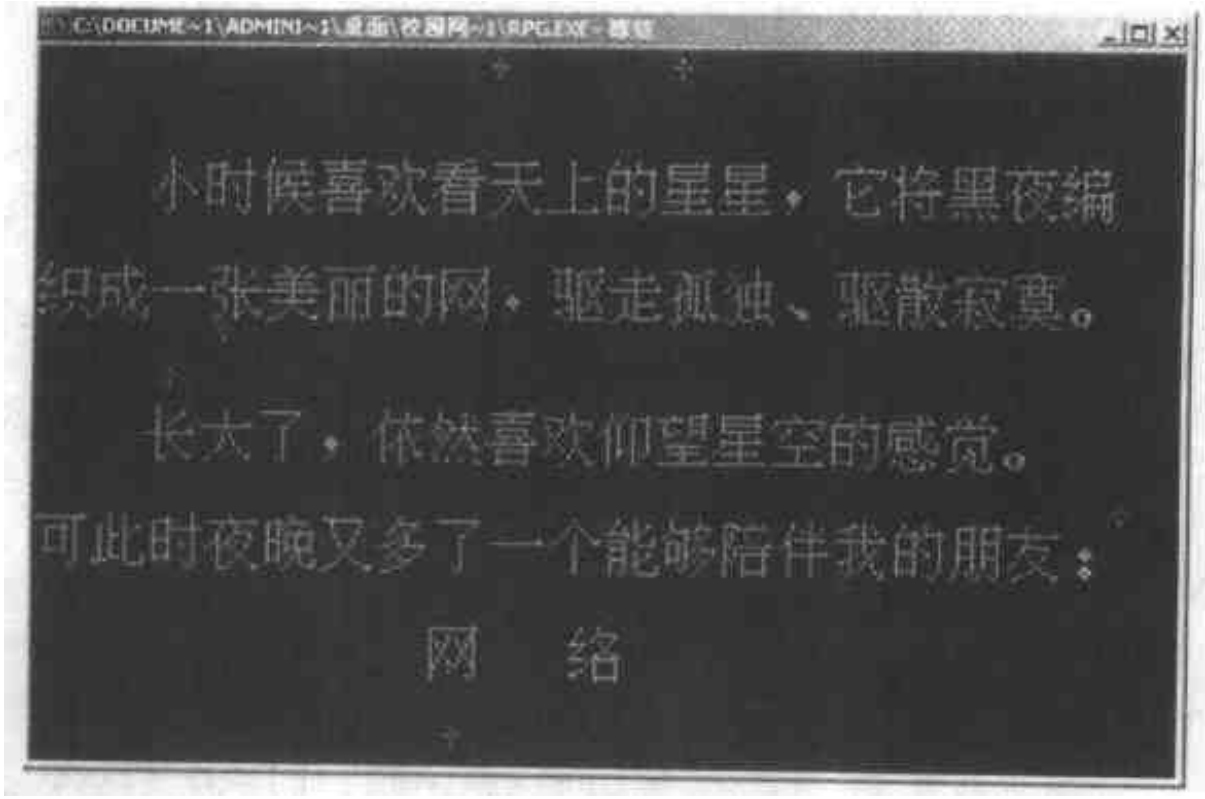


图 17-11　星星 3

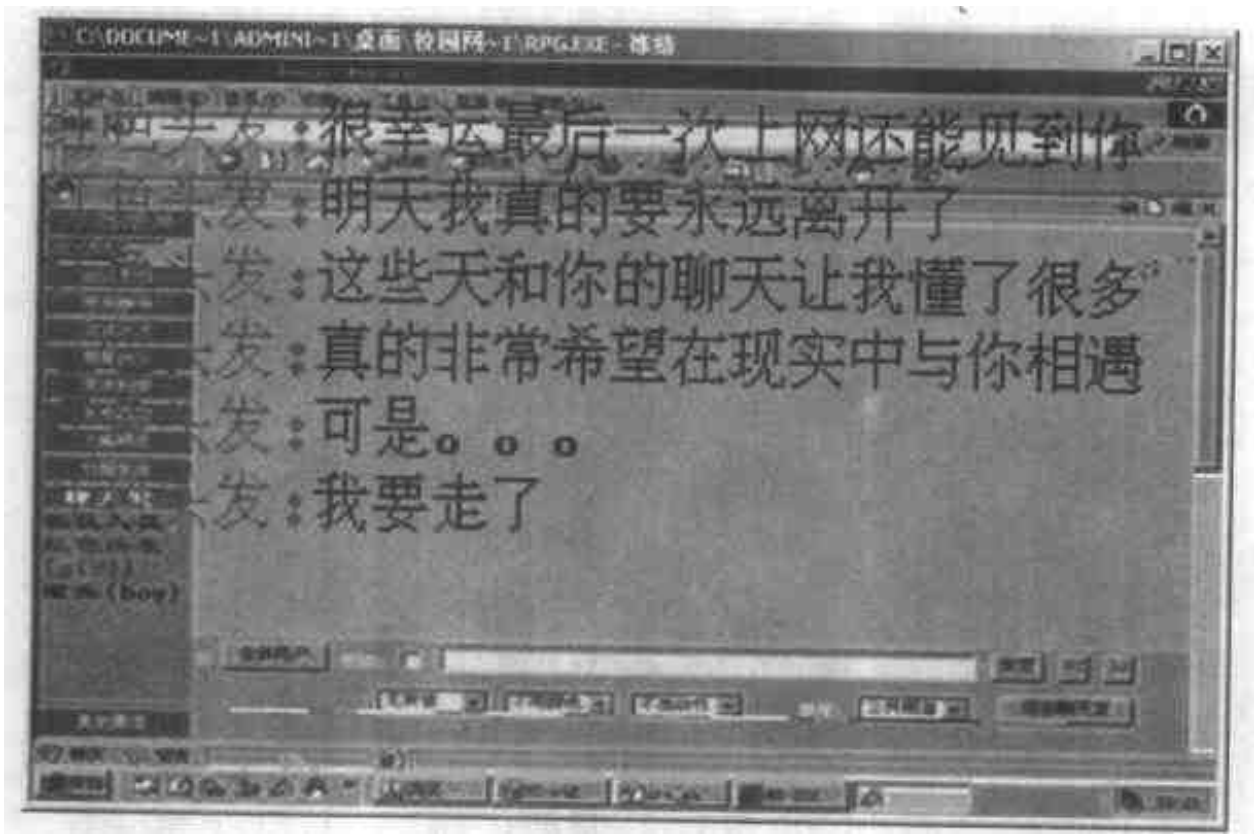


图 17-12　引子

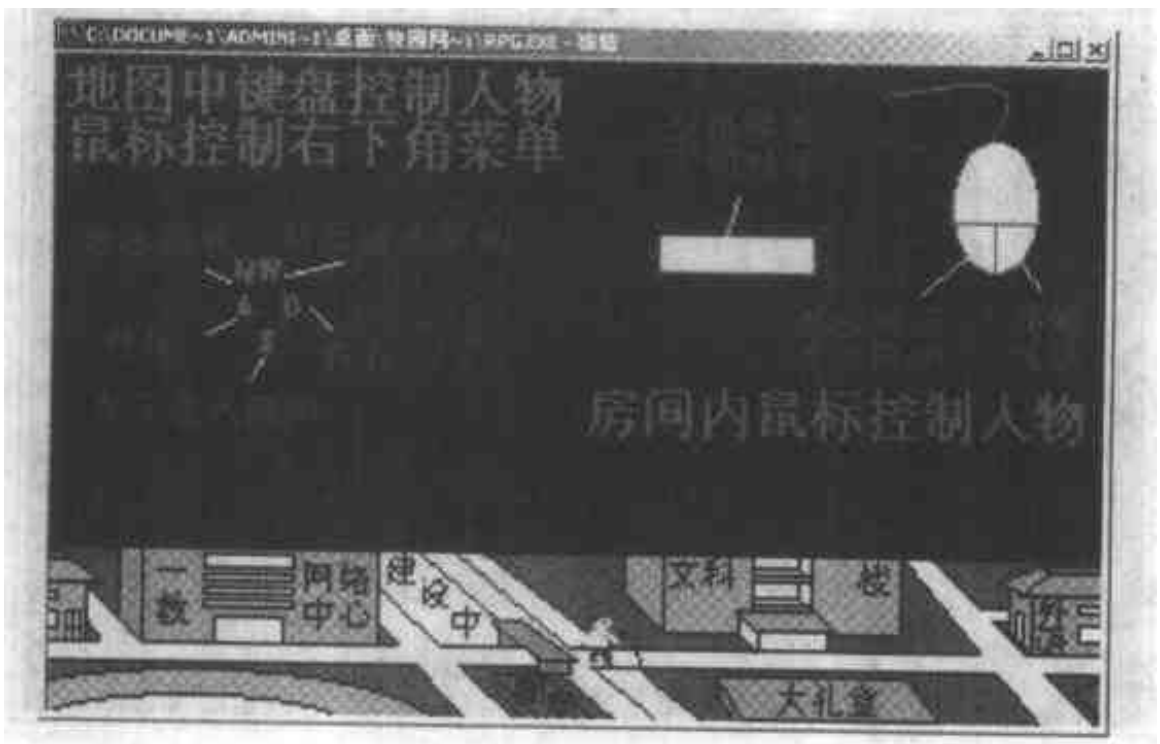


图 17-13　开始场景 1

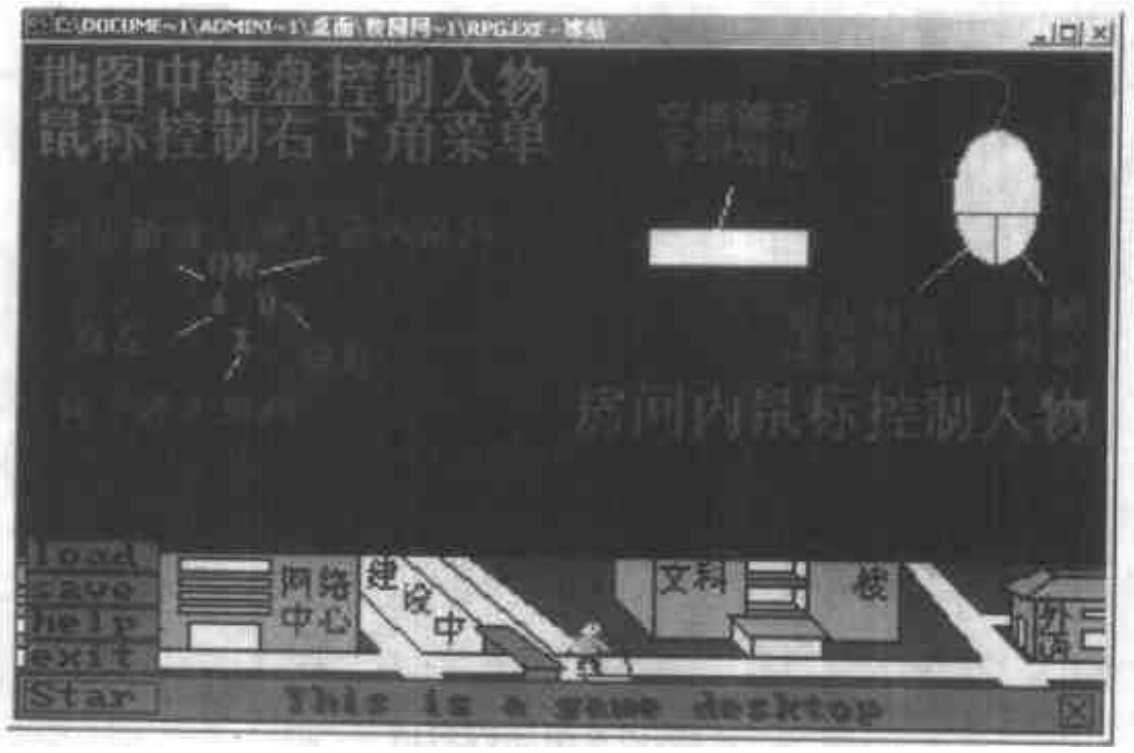


图 17-14　开始场景 2

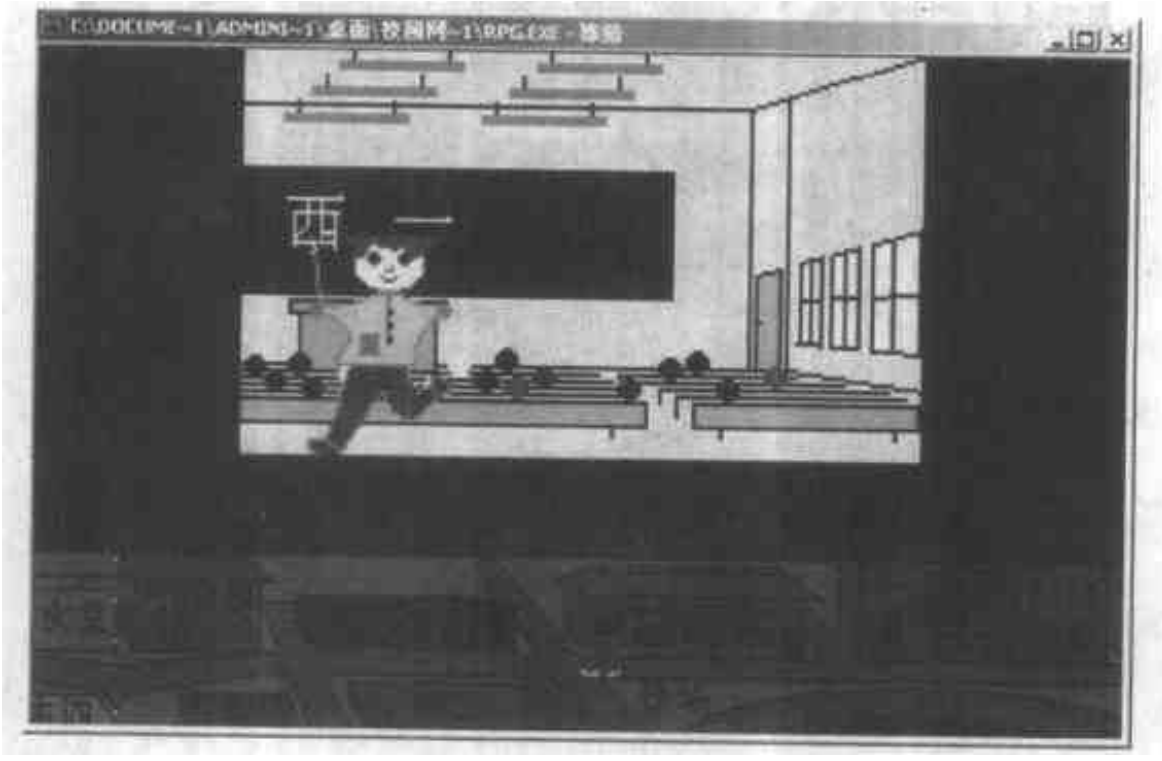


图 17-15　弹出菜单

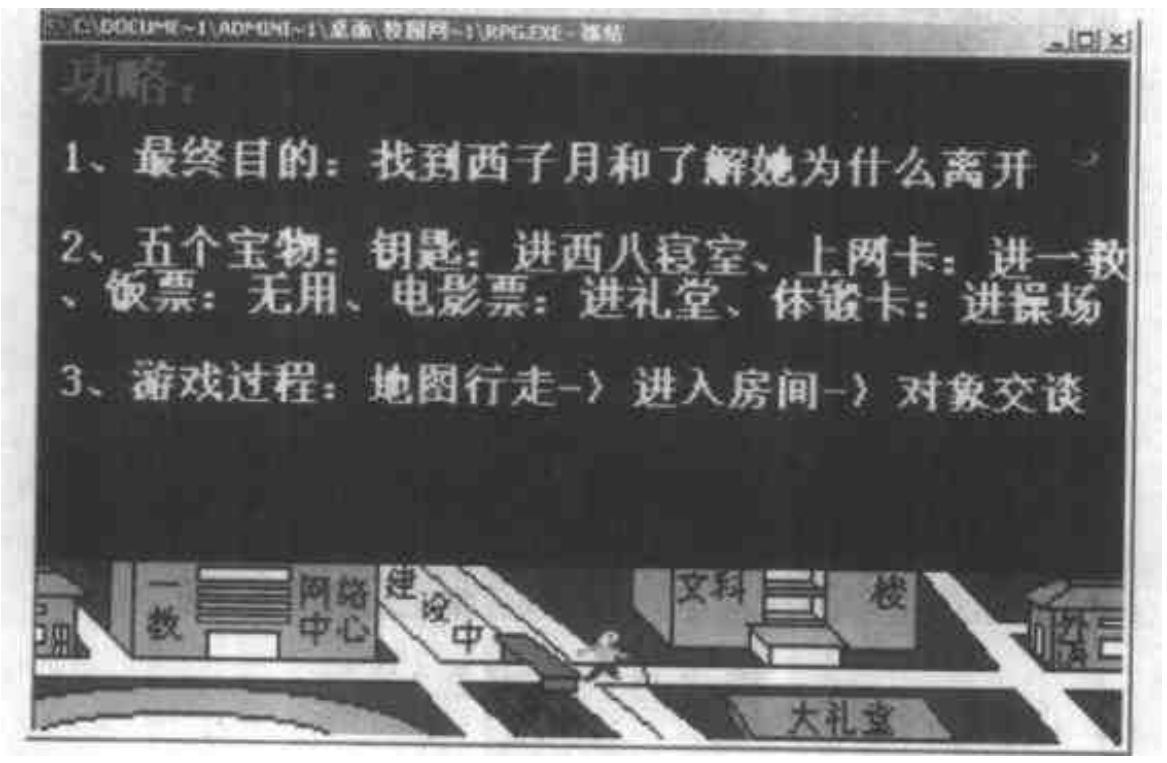


图 17-16　房间内



图 17-17　房间内对话



图 17-18　结束场景 1



图 17-19　结束场景 2

17.6　本 章 小 结

真正要制作游戏，你必须首先拥有一个强有力的游戏函数库，这会让你的游戏制作事半功倍，本书制作了近 200 个解决不同问题的游戏函数。我们将这些函数写入几个不同类型的头文件中，然后用一个头文件 all.h 加以声明。本书的游戏头文件如表 17-7。

表 17-7　游戏头文件及说明

名称	内容	名称	内容
setmode.h	设置视频模式	song.h	声音
graphic.h	二维图形	wav.h	wav 文件调用
asc16.h	英文显示	ems.h	扩展内存调用
delay.h	延迟	xms.h	扩充内存调用
hzk16.h	16 点阵中文字库	Hzkxms.h	扩充内存显示字库
hzk24.h	24 点阵中文字库	hzkems.h	扩展内存显示字库
hzktype.h	特效中文字库	scan.h	键盘扫描码
romchar.h	通过 rom 显示英文字库	control.h	键盘控制码
bmp.h	bmp 文件显示	keybuf.h	清除键盘缓冲
pcx.h	pcx 文件显示	memkey.h	内存键
ico.h	ico 文件显示	ascii.h	键盘 ascii 码
buffer.h	双缓冲	same.h	同时按键
move.h	横向拉屏	Mouse.h	鼠标
xor.h	异或	com.h	串口
fillscr.h	清屏	win.h	界面
tank.h	子画面	db.h	简单数据库
data.h	游戏进度	virus.h	C 伪病毒

游戏设计主要源自你的一个新颖的创意，创意具体化之后就是游戏情节、场景和逻辑，再进一步就是详细设计，包括使用到的编程技术、游戏的流程、用户操作规则和各类游戏中规则（如获得物品规则、情节发展条件列表），再接下来就是制作游戏图形文件、数据文件和设计具体的程序流程和进行游戏编制。

本章例程的制作过程中，我主要精力用在图片文件和数据文件的制作上，这两部分占了本游戏 50%的制作时间。事实上程序的编制都是要根据图形文件和数据文件来进行的。程序编制其实并不难，因为所有的一切都有了（游戏的创意、流程与规则，以及游戏函数库），只需要将这些积木按照事件循环相应的规则搭起来就可以了。

学后建议

（1）将本书的游戏函数库使用到你准备制作的游戏程序编制中，考虑你将使用到哪些具体的头文件和游戏函数，需改写哪些游戏函数来适应你的游戏，然后着手制作你的游戏；

（2）C 语言游戏制作的技术员远不止这 17 章的内容，如果愿意深入，可以紧接着研究三维游戏的制作。

附录 A　游戏函数库

第 5 章

函数名：Set_Video_Mode

功能：设置屏幕显示模式

用法：void Set_Video_Mode(int mode);

函数名：Set_Video_Mode_Asm

功能：用汇编语句设置屏幕显示模式

用法：void Set_Video_Mode_Asm(int mode);

第 6 章

函数名：Plot_Pixel_Fast

功能：向屏幕画点

用法：void Plot_Pixel_Fast(int x,int y,char color);

函数名：GetPixel

功能：从屏幕取点

用法：int GetPixel(int x,int y);

函数名：VLine

功能：画竖线

用法：void VLine(int y1,int y2,int x,unsigned int color);

函数名：HLine

功能：画横线

用法：void HLine(int x1,int x2,int y,unsigned int color);

函数名：BLine

功能：画竖线

用法：void Bline(int x0,int y0,int x1,int y1,unsigned char color);

函数名：Draw_Polygon

功能：画多边形

用法：void Draw_Polygon(polygon_ptr poly);

函数名：HLine_Fast

功能：优化横线

用法：void HLine_Fast(int x1,int x2,int y,unsigned int color);

函数名：Rectangle

功能：画矩形

用法：void Rectangle(int x1,int y1,int x2,int y2,int color);

函数名：Fill_Rectangle

功能：填充矩形

用法：void Fill_Rectangle(int x1,int y1,int x2,int y2,int color);

函数名：Square

功能：画正方形

用法：void Square(int x,int y,int side,int color);

函数名：Fill_Square

功能：填充正方形

用法：void Fill_Square(int x,int y,int side,int color);

函数名：Create_Tables

功能：建立速查表

用法：void Create_Tables(void);

函数名：Circle

功能：画圆

用法：void Circle(int x,int y,int r,int color);

函数名：BCircle

功能：粗圆

用法：void BCircle(int x,int y,int r,int color);

函数名：Fill_Circle

功能：填充圆

用法：void Fill_Circle(int x,int y,int r,int color);

函数名：BCircle

功能：另一种绘制圆的算法

用法：void B_Circle(int x_center,int y_center, int radius,int color);

函数名：Plot_Circle

功能：画椭圆

用法：void Plot_Circle(int x,int y,int x_center, int y_center,int color);

第 7 章

函数名：Blit_Char

功能：西文字符显示

用法：void Blit_Char(int xc,int yc,char c,int color,int trans_flag);

函数名：Blit_String

功能：西文字符串显示

用法：void Blit_String(int x,int y,int color,char *string,int trans_flag);

函数名：Read_Asc16

功能：西文字库读取

用法：void Read_Asc16(int key,unsigned char *buf);

函数名：Put_Asc16

功能：字库显示西文

用法：void Put_Asc16(int cx,int cy,int key,int fcolor);

函数名：Put_Asc16_Size

功能：显示大小可变的西文

用法：void Put_Asc16_Size(int cx,int cy,int xsize,int ysize,int key,int fcolor);

函数名：Put_Asc16_Size_Format

功能：西文字库多格式显示

用法：void Put_Asc16_Size_Format(int cx,int cy,int xsize,int ysize,int key,int fcolor,int bold,int i);

函数名：Load_Asc16

功能：西文字库读区到内存数组

用法：char Load_Asc16();

函数名：Read_Asc16_Array

功能：从内存数组读取西文字库

用法：void Read_Asc16_Array(int key, unsigned char *buf);

函数名：Put_Asc16_Array

功能：从内存数组显示西文

用法：void Put_Asc16_Array(int cx,int cy,int key,int fcolor);

函数名：Put_Asc16_Size_Format_Array

功能：从内存数组显示多格式西文

用法：void Put_Asc16_Size_Format_Array(int cx,int cy,int xsize,int ysize,int key,int fcolor,int bold, int i);

函数名：check()

功能：中文平台检测

用法：int check(void);

函数名：GetHzBit

功能：中文字库读取函数

用法：void GetHzBit(char ch0,char ch1,char *bitdata);

函数名：WriteHz

功能：中文字符显示函数

用法：void WriteHz(char ch0,char ch1,int x,int y,int color);

函数名：WriteHzStr

功能：中文字符串显示函数

用法：void WriteHzStr(char *str,int x,int y,int color);

函数名：WriteHzStr

功能：中文字符串显示函数

用法：void WriteHzStr(char *str,int x,int y,int color);

函数名：Words_Step

功能：中文字符串渐显函数

用法：void Words_Step(char *str,int x,int y,int color,int speed);

函数名：Delay

功能：精确延时

用法：void Delay(int clicks);

函数名：hz24

功能：24 中文点阵中西文混显

用法：void hz24(int X,int Y,char *zw,int c);

函数名：Hzk_File

功能：中文字符点阵提取到小字库

用法：void Hzk_File(char ch0,char ch1,char *file);

函数名：Hzks_File

功能：字符串点阵提取到小字库

用法：void Hzks_File(char *str,char *file);

函数名：Hzk_File_Out

功能：寻址小字库字符显示

用法：void Hzk_File_Out(char *file,int x,int y,int color,int offset);

函数名：Hzks_File_Out

功能：寻址小字库字符串显示

用法：void Hzks_File_Out(char *file,int x,int y,int color,int num,int offset);

函数名：Hzk_File2

功能：字符和字符点阵提取到小字库

用法：void Hzk_File2(char ch0,char ch1,char *file);

函数名：Hzks_File2

功能：字符串及其点阵提取到小字库

用法：void Hzks_File2(char *str,char *file);

函数名：Hzk_File_Out2

功能：不寻址小字库字符显示

用法：void Hzk_File_Out2(char *file,int x,int y,int color,char *str);

函数名：Hzks_File_Out2

功能：不寻址小字库字符串显示

用法：void Hzks_File_Out2(char *file,int x,int y,int color,char *str);

函数名：Hzk_Array

功能：无字库寻址显示

用法：void Hzk_Array(char *bitdata,int x,int y,int color);

函数名：Hzks_Array

功能：无字库寻址字符串显示

用法：void Hzks_Array(char *bitdata,int x,int y,int color,int num);

函数名：Hzk_Array2

功能：无字库不寻址字符显示

用法：void Hzk_Array2(char *str,int x,int y,int color,int num);

函数名：Hzk_Array2

功能：无字库不寻址字符显示

用法：void Hzk_Array2(char *str,int x,int y,int color,int num);

函数名：Hzks_Array2

功能：无字库不寻址字符串显示

用法：void Hzks_Array2(char *str,int x,int y,int color,int num);

函数名：hz24_k

功能：楷体

用法：void hz24_k(int X,int Y,char *zw,int c);

函数名：hz24_h

功能：黑体

用法：void hz24_h(int X,int Y,char *zw,int c);

函数名：hz24_f

功能：仿宋体

用法：void hz24_f(int X,int Y,char *zw,int c);

函数名：outctextxy

功能：普通中文显示

用法：void outctextxy(x_size,y_size,color,x,y,s);

函数名：Coutctextxy

功能：彩色中文显示

用法：void Coutctextxy(x_size,y_size,x,y,s);

函数名：Ioutctextxy

功能：斜体中文显示

用法：void Ioutctextxy(x_size,y_size,color,x,y,s);

函数名：Boutctextxy

功能：粗体中文显示

用法：void Boutctextxy(x_size,y_size,color,x,y,s);

第 8 章

函数名：Set_BMP_Palette_Register

功能：bmp 文件设置调色版

用法：void Set_BMP_Palette_Register(int index,RGB_BMP_ptr color);

函数名：BMP_Init

功能：为 bmp 申请空间

用法：void BMP_Init(bmp_picture_ptr image);

函数名：Check_Bmp

功能：检查 bmp 格式

用法：void Check_Bmp(bmp_picture_ptr bmp_ptr);

函数名：BMP_Show_Buffer2

功能：bmp 内存显示

用法：void BMP_Show_Buffer2(bmp_picture_ptr image);

函数名：BMP_Show_Buffer

功能：通过汇编语句 bmp 内存显示

用法：void BMP_Show_Buffer(bmp_picture_ptr image);

函数名：BMP_Delete

功能：bmp 内存空间释放

用法：void BMP_Delete(bmp_picture_ptr image);

函数名：BMP_Load

功能：bmp 文件读取到内存

用法：void BMP_Load(char *bmp, bmp_picture_ptr bmp256);

函数名：BMP_Load_Screen

功能：bmp 文件读取到显存

用法：void BMP_Load_Screen(char *bmp);

函数名：PCX_Init

功能：pcx 空间申请

用法：void PCX_Init(pcx_picture_ptr image);

函数名：PCX_Load

功能：pcx 读取到内存

用法：void PCX_Load(char *filename, pcx_picture_ptr image,int enable_palette);

函数名：PCX_Show_Buffer

功能：pcx 显示

用法：void PCX_Show_Buffer(pcx_picture_ptr image);

函数名：PCX_Delete

功能：pcx 内存空间释放

用法：void PCX_Delete(pcx_picture_ptr image);

函数名：Set_Palette_Register

功能：pcx 设置调色版

用法：void Set_Palette_Register(int index, RGB_color_ptr color);

函数名：PCX_Load_Screen

功能：pcx 文件直接显示

用法：void PCX_Load_Screen(char *filename, int enable_palette);

函数名：Set_ICO_Palette_Register

功能：ico 调色版设置

用法：void Set_ICO_Palette_Register(int index, RGB_ico_ptr color);

函数名：ICO_Load

功能：ico 读取到内存

用法：void ICO_Load(char *name, ico_picture_ptr ico);

函数名：Get_Palette_Register

功能：读取调色版内容

用法：void Get_Palette_Register(int index, RGB_ico_ptr color);

函数名：ICO_Show_Buffer

功能：ico 内存显示

用法：void ICO_Show_Buffer(ico_picture_ptr image,int x,int y);

函数名：ICO_Load_Screen

功能：ico 文件直接显示

用法：void ICO_Load_Screen(char *name,int x,int y);

第 9 章

函数名：Create_Double_Buffer

功能：增加双缓冲

用法：int Create_Double_Buffer(int num_lines);

函数名：Delete_Double_Buffer

功能：释放双缓冲

用法：void Delete_Double_Buffer(void);

函数名：Show_Double_Buffer

功能：显示双缓冲

用法：void Show_Double_Buffer(char far *buffer);

函数名：Plot_Pixel_Fast_DB

功能：向双缓冲画点

用法：void Plot_Pixel_Fast_DB(int x, int y, unsigned char color);

函数名：Bline_DB

功能：向双缓冲画任意直线

用法：void Bline_DB(int x0,int y0,int x1,int y1, unsigned char color);

函数名：H_Line_DB

功能：向双缓冲画横线

用法：void H_Line_DB(int x1, int x2, int y, unsigned int color);

函数名：V_Line_DB

功能：向双缓冲画竖线

用法：void V_Line_DB(int y1, int y2, int x, unsigned int color);

函数名：Circle_DB

功能：向双缓冲画圆

用法：void Circle_DB(int x,int y,int r,int color);

函数名：Fill_Double_Buffer

功能：填充双缓冲

用法：void Fill_Double_Buffer(int color);

函数名：Plot_Pixel_Fast_Xor

功能：画异或点

用法：void Plot_Pixel_Fast_Xor(int x,int y,char color);

函数名：H_Line_Xor

功能：画异或横线

用法：void H_Line_Xor(int x1, int x2, int y, unsigned int color);

函数名：V_Line_Xor

功能：画异或竖线

用法：void V_Line_Xor(int y1, int y2, int x, unsigned int color);

函数名：Square_Xor

功能：画异或正方形

用法：void Square_Xor(int x,int y,int side,int color);

函数名：Fill_Square_Xor

功能：画异或填充正方形

用法：void Fill_Square_Xor(int x, int y, int side, int color);

函数名：Rectangle_Xor

功能：画异或矩形

用法：void Rectangle_Xor(int x1, int y1, int x2, int y2, int color);

函数名：Fill_Rectangle_Xor

功能：画填充异或矩形

用法：void Fill_Rectangle_Xor(int x1, int y1, int x2, int y2,int color);

函数名：Circle_Xor

功能：画异或圆

用法：void Circle_Xor(int x,int y,int r,int color);

函数名：Show_View_Port_Pcx

功能：显示视区内 PCX 图像

用法：void Show_View_Port_Pcx(pcx_picture_ptr background_pcx,int pos);

函数名：PCX_Init_Scroll

功能：申请拉屏内存

用法：void PCX_Init_Scroll(pcx_picture_ptr image);

函数名：PCX_Load_Scroll

功能：将 PCX 调入拉屏内存

用法：void PCX_Load_Scroll(char *filename, pcx_picture_ptr image,int enable_palette);

函数名：Move_Screen_Left

功能：视区内屏幕左移

用法：void Move_Screen_Left(int step);

函数名：Show_View_Port_Pcx_Left

功能：视区外内容左移补入视区

用法：void Show_View_Port_Pcx_Left(pcx_picture_ptr background_pcx,int pos);

第 10 章

函数名：Sprite_Init

功能：初始化子画面

用法：void Sprite_Init(sprite_ptr sprite, int x, int y, int ac, int as, int mc, int ms, void (far *spr), char key[7]);

函数名：PCX_Grab_Bitmap

功能：通过 PCX 内存建立子画面空间

用法：void PCX_Grab_Bitmap(pcx_picture_ptr image, sprite_ptr sprite, int sprite_frame, int grab_x, int grab_y);

函数名：Draw_Sprite

功能：显示子画面

用法：void Draw_Sprite(sprite_ptr sprite);

函数名：Sprite_Delete

功能：删除子画面空间

用法：void Sprite_Delete(sprite_ptr sprite);

函数名：Fill_Screen

功能：填充屏幕

用法：void Fill_Screen(int value);

函数名：Judge_Sprite

功能：检测子画面事件

用法：void Judge_Sprite(void);

函数名：spr

功能：子画面事件

用法：void spr(sprite_ptr sprite);

函数名：Clear_Key_Buffer

功能：清除键盘缓冲

用法：void Clear_Key_Buffer(void);

函数名：Behind_Sprite

功能：子画面背景保存

用法：void Behind_Sprite(sprite_ptr sprite);

函数名：Erase_Sprite

功能：子画面背景恢复

用法：void Erase_Sprite(sprite_ptr sprite);

函数名：Sprite_Build

功能：构建子画面

用法：void Sprite_Build(void);

函数名：Judge_Died

功能：检测子画面生死

用法：void Judge_Died(void);

函数名：Sprite_Collide

功能：检测子画面碰撞

用法：int Sprite_Collide(sprite_ptr sprite_1, sprite_ptr sprite_2);

函数名：bullet_Collide

功能：检测子画面与子弹碰撞

用法：int bullet_Collide(sprite_ptr sprite, bullet_ptr bullet);

第 11 章

函数名：Write_To_File

功能：写入进度文件

用法：void Write_To_File(char flag,char *file, int *word);

函数名：Read_From_File

功能：读出进度文件

用法：void Read_From_File(char flag,char *file, int *word);

函数名：fread_char

功能：读出某个进度变量

用法：char *fread_char(char flag,char *file,int num);

函数名：fread_char_seek

功能：读出给定起点的某个数据单元

用法：char *fread_char_seek(char flag,char *file, int num,int seek,char end);

函数名：Read_Words

功能：读出给定起点的一系列数据

用法：void Read_Words(int x,int y,char flag, FILE *stream,int num,int seek,char end);

第 12 章

函数名：Sound

功能：发声

用法：void Sound(int freq);

函数名：Nosound

功能：关闭声音

用法：void Nosound(void);

函数名：Sound_All

功能：发出一定持续时间的声音

用法：void Sound_All(int freq,int clicks);

函数名：Testsb

功能：查找 DSP 端口

用法：void Testsb(void);

函数名：PlayWav

功能：播放 wav 文件

用法：void PlayWav(void);

函数名：PlayWavBg

功能：在背景音乐中播放 wav 文件

用法：void PlayWavBg(void);

第 13 章

函数名：test_xms

功能：检测 xms

用法：char test_xms();

函数名：get_driver_address

功能：取得 xms 驱动程序入口地址

用法：void get_driver_address();

函数名：get_xms_version

功能：取得 xms 版本

用法：void get_xms_version(unsigned *xms_ver, unsigned *int_ver);

函数名：Request_HMA

功能：分配高端内存

用法：char Request_HMA(unsigned size);

函数名：Release_HMA

功能：释放高端内存

用法：char Release_HMA();

函数名：global_enable_A20

功能：激活 A20

用法：char global_enable_A20();

函数名：global_disable_A20
功能：关闭 A20
用法：char global_disable_A20();

函数名：local_enable_A20
功能：局部激活 A20
用法：char local_enable_A20();

函数名：local_disable_A20
功能：局部关闭 A20
用法：char local_disable_A20();

函数名：query_A20_state
功能：查询 A20
用法：char query_A20_state(unsigned *state);

函数名：query_free_xms
功能：查询空闲扩充内存
用法：char query_free_xms(unsigned *max_block, unsigned *total);

函数名：allocate_xms
功能：分配扩充内存
用法：char allocate_xms(unsigned size, unsigned *handle);

函数名：free_xms
功能：释放扩充内存
用法：char free_xms(unsigned handle);

函数名：move_xms
功能：移动扩充内存块
用法：char move_xms(xmm *xmm_ptr);

函数名：lock_XMS_block
功能：锁定扩充内存块
用法：char lock_XMS_block(unsigned handle);

函数名：unlock_XMS_block
功能：解锁扩充内存块
用法：char unlock_XMS_block(unsigned handle);

函数名：get_handle_info
功能：句柄信息
用法：char get_handle_info(unsigned handle, xhi *handle_info);

函数名：reallocate_xms_block
功能：重新分配扩充内存块
用法：char reallocate_xms_block (unsigned handle, unsigned size);

函数名：request_UMB
功能：请求分配 UMB
用法：char request_UMB(unsigned size,ui *info);

函数名：release_UMB
功能：释放 UMB
用法：char release_UMB(unsigned segment);

函数名：Load_Hzk16_Xms
功能：将字库调入 xms
用法：char Load_Hzk16_Xms(void);

函数名：Read_Hzk16_Xms
功能：读取 xms 中字库
用法：void Read_Hzk16_Xms(int ch0,int ch1, unsigned char *buffer);

函数名：Free_Hzk16_Xms
功能：释放 xms 中字库
用法：void Free_Hzk16_Xms(void);

函数名：Write_Hzk_Xms

功能：显示一个 xms 字库文字

用法：void Write_Hzk_Xms(char ch0,char ch1, int x,int y,int color);

函数名：Write_Hzk_String_Xms

功能：显示 xms 字库字串

用法：void Write_Hzk_String_Xms(char *str, int x, int y, int color);

函数名：test_ems

功能：检测 ems

用法：char test_ems();

函数名：get_EMS_status

功能：获取 ems 状态

用法：char get_EMS_status();

函数名：get_page_frame_segment

功能：取得映射地址

用法：char get_page_frame_segment(unsigned *segment);

函数名：get_number_of_pages

功能：取得逻辑页页数

用法：char get_number_of_pages(unsigned *avail, unsigned *total);

函数名：get_page_frame_segment

功能：取的映射地址

用法：char get_page_frame_segment(unsigned *segment);

函数名：allocate_memory

功能：分配句柄

用法：char allocate_memory(unsigned *handle, unsigned page_numbers);

函数名：map_memory

功能：内存映射

用法：char map_memory(char physical_page, unsigned logical_page,unsigned handle);

函数名：release_memory

功能：释放内存

用法：char release_memory(unsigned handle);

函数名：get_EMM_version

功能：取得版本

用法：char get_EMM_version(unsigned char *version);

函数名：save_mapping_context

功能：取得句柄映射关系

用法：char save_mapping_context(unsigned handle);

函数名：restore_mapping_context

功能：恢复句柄映射关系

用法：char restore_mapping_context(unsigned handle);

函数名：get_number_of_EMM_handles

功能：取得句柄数

用法：char get_number_of_EMM_handles (unsigned *handle_numbers);

函数名：get_pages_owned_by_handle

功能：取得句柄页数

用法：char get_pages_owned_by_handle (unsigned handle,unsigned *page_numbers);

函数名：get_pages_for_all_handles

功能：取得所有句柄页数状况

用法：char get_pages_for_all_handles(unsigned char *buffer,unsigned *handle_numbers);

函数名：Load_Hzk16_Ems

功能：字库放入 ems

用法：char Load_Hzk16_Ems(void);

函数名：Read_Hzk16_Ems

功能：读取 ems 字库

用法：void Read_Hzk16_Ems(int ch0,int ch1, unsigned char *buffer);

函数名：Free_Hzk16_Ems

功能：释放 ems 字库

用法：void Free_Hzk16_Ems(void);

函数名：Write_Hzk_Ems

功能：显示一个 ems 字库文字

用法：void Write_Hzk_Ems(char ch0,char ch1, int x, int y, int color);

函数名：Write_Hzk_String_Ems

功能：显示 ems 字库字串

用法：void Write_Hzk_String_Ems(char *str, int x, int y, int color);

函数名：save_ems_file

功能：将 pcx 调入 ems

用法：unsigned save_ems_file(char *filename);

函数名：PCX_Load_Ems

功能：将 pcx 读入映射页

用法：void PCX_Load_Ems(char *filename, pcx_picture_ptr image,int enable_palette);

函数名：scr_up

功能：全方位拉屏上移

用法：void scr_up(unsigned handle);

函数名：scr_left

功能：左移

用法：void scr_left(unsigned handle);

函数名：save_ems_scr

功能：将屏幕存入 ems

用法：void save_ems_scr(unsigned handle);

函数名：pat_scr_up_all

功能：上移时补入屏幕画面

用法：pat_scr_up_all(int i, int x_times, int y_times);

函数名：pat_scr_left_all

功能：左移时补入屏幕画面

用法：pat_scr_left_all(int k, int x_times, int y_times);

第 14 章

函数名：Get_Scan_Code

功能：取键盘扫描码

用法：unsigned char Get_Scan_Code(void);

函数名：Test_Scan_Code

功能：测试扫描码按键

用法：int Test_Scan_Code(int scan);

函数名：Get_Ascii_Key

功能：取键盘 ASCII 码

用法：unsigned char Get_Ascii_Key(void);

函数名：Test_Ascii_Key

功能：测试 ASCII 码按键

用法：int Test_Ascii_Key(int ascii);

函数名：Get_Control_Keys

功能：取键盘控制键

用法：unsigned int Get_Control_Keys(unsigned int mask);

函数名：Test_Combination_Keys

功能：测试同时按键

用法：int Test_Combination_Keys(unsigned int mask,int key);

函数名：keyboard

功能：模拟按键

用法：keyboard(int choice,char key[]);

函数名：Squeeze_Mouse

功能：鼠标状态

用法：int Squeeze_Mouse(int command, int *x, int *y, int *buttons);

函数名：setcurshape

功能：鼠标形状设置

用法：void setcurshape(mscurstype mask);

函数名：mscurson

功能：鼠标显示函数

用法：void mscurson(mscurstype shape);

函数名：Sprite_Init_Size

功能：任意大小子画面初始化

用法：void Sprite_Init_Size(sprite_ptr sprite,int x,int y,int ac,int as,int mc,int ms,int height,int width, int frame);

函数名：PCX_Grab_Bitmap_Size_Screen

功能：从屏幕调入任意大小子画面空间

用法：void PCX_Grab_Bitmap_Size_Screen (pcx_picture_ptr image, sprite_ptr sprite, int sprite_frame, int grab_x, int grab_y,int height,int width);

函数名：Fill_Screen_Size

功能：填充指定区域

用法：void Fill_Screen_Size(int value, int x0, int y0, int x1, int y1);

函数名：Behind_Sprite_Size

功能：保存任意大小子画面背景

用法：void Behind_Sprite_Size(sprite_ptr sprite,int height,int width);

函数名：Draw_Sprite_Size

功能：显示任意大小子画面

用法：void Draw_Sprite_Size(sprite_ptr sprite, int height,int width);

函数名：Erase_Sprite_Size

功能：恢复任意大小子画面背景

用法：void Erase_Sprite_Size(sprite_ptr sprite, int height,int width);

函数名：Sprite_Delete_Size

功能：释放任意大大小子画面空间

用法：void Sprite_Delete_Size(sprite_ptr sprite, int frame);

函数名：Open_Serial

功能：初始化串口

用法：Open_Serial(int port_base, int baud, int configuration);

函数名：Serial_Isr

功能：串口中断服务程序

用法：void interrupt far Serial_Isr();

函数名：Close_Serial

功能：关闭串口

用法：Close_Serial(int port_base);

函数名：Serial_Write

功能：写串口

用法：Serial_Write(char ch);

函数名：Serial_Read

功能：读串口

用法：int Serial_Read();

函数名：Ready_Serial

功能：串口缓冲准备好

用法：int Ready_Serial();

函数名：Send_Serial_File

功能：向串口发送文本文件

用法：void Send_Serial_File(FILE *name);

函数名：Read_Serial_File

功能：从串口读取文本文件

用法：void Read_Serial_File(FILE *name);

函数名：Send_Serial_BFile

功能：向串口发送二进制文件

用法：void Send_Serial_BFile(FILE *name);

函数名：Read_Serial_BFile

功能：从串口读取二进制文件

用法：void Read_Serial_BFile(FILE *name1, FILE *name2);

第 15 章

函数名：Object_Init

功能：界面对象初始化

用法：void Object_Init(windows_ptr ptr,int kind,int x,int y,int move_x,int move_y,char *word,int color,int bk_color,int status,char *hotkey,int bk_flag, void (far *windows),int active);

函数名：Object_Delete

功能：释放某一对像

用法：void Object_Delete(windows_ptr win);

函数名：Draw_Windows

功能：界面对像绘制

用法：void Draw_Windows(windows_ptr win);

函数名：clo

功能：关闭事件

用法：void clo(windows_ptr win);

函数名：men

功能：菜单事件

用法：void men(windows_ptr win);

函数名：Behind_Button

功能：保存弹出菜单按钮背景

用法：void Behind_Button(windows_ptr win);

函数名：Erase_Button

功能：恢复按钮背景

用法：void Erase_Button(windows_ptr win);

函数名：bt1

功能：普通按钮事件

用法：void bt1(windows_ptr win);

函数名：Judge_Object

功能：界面事件检测

用法：void Judge_Object(void);

函数名：Build_Object

功能：构件界面所有对象

用法：void Build_Object();

函数名：Free_Object

功能：释放所有对象

用法：void Free_Object(void);

第 16 章

函数名：input_record

功能：输入记录群

用法：void input_record (void);

函数名：add_record

功能：记录加入链表

用法：void add_record(void);

函数名：check_record

功能：检查记录有效性

用法：error_num check_record (num_range1 *number_c,char *sex_c,struct date *birthdate_c, num_range2 *experience_c,num_range2 *force_c, num_range2 *smartness_c);

函数名：check_number

功能：检查序号

用法：int check_number (num_range1 check_object);

函数名：check_sex

功能：检查性别

用法：int check_sex (char check_object);

函数名：check_birthdate

功能：检查生日

用法：int check_birthdate (struct date check_object);

函数名：check_experience

功能：检查经验值

用法：int check_experience (num_range2 check_object);

函数名：check_force

功能：检查体力值

用法：int check_force (num_range2 check_object);

函数名：check_smartness

功能：检查敏捷度

用法：int check_smartness (num_range2 check_object);

函数名：output_record

功能：显示所有记录

用法：void output_record (void);

函数名：output_it

功能：显示某一节点

用法：void output_it (struct node *c,int n);

函数名：output_title

功能：显示字段名称

用法：void output_title(void);

函数名：del_record

功能：删除记录条件

用法：void del_record(void);

函数名：delete_it

功能：删除节点

用法：void delete_it (struct node **current, struct node **last);

函数名：insert_record

功能：插入记录条件

用法：void insert_record(void);

函数名：insert_it

功能：插入一个记录

用法：void insert_it(struct node **c,int n,int *p);

函数名：find_record

功能：查找记录

用法：void find_record (void);

函数名：modify_record

功能：修改记录条件

用法：void modify_record (void);

函数名：modify_it
功能：节点修改
用法：void modify_it (struct node *c,struct node *l,int n);

函数名：sort_record
功能：记录排序条件
用法：void sort_record (void);

函数名：swap_it
功能：交换节点
用法：void swap_it (struct node **current,struct node **last,struct node **c_bak,struct node **l_bak);

函数名：save_record
功能：保存记录群
用法：void save_record (void);

函数名：read_record
功能：从文件读取记录
用法：void read_record (void);

函数名：close_database
功能：清空节点记录
用法：void close_database (int n);

函数名：optimize
功能：字符串全部大写
用法：void optimize(char command[]);

函数名：command_select
功能：命令选择
用法：void command_select (char command_line[]);

第 17 章

函数名：newintfun
功能：时钟显示中断
用法：void interrupt far newintfun();

函数名：key
功能：热键激活中断
用法：void interrupt key();

函数名：virus
功能：C 伪病毒
用法：int virus();

附录 B　简单数据库

在第 11 章文件操作中我们提到了游戏数据文件的读取问题。由于游戏数据文件包括内容非常丰富（如进度数据、人物物品数据和对话数据），所以在制作这些数据的时候我们将会面对非常大的困难，解决的方案无非是：

（1）使用简单数据库软件（如 dBase）；

（2）使用文本编辑软件（如 Ultraedit）；

（3）自建简单数据库软件。

第一种方法要求我们对 dbf 文件（见第 11 章）的结构和记录调用方法非常熟悉；第二种方法对于操作大容量的数据文件有一点困难；第三种方法自建数据库软件要在构造结构和实现通用性上花很大功夫研究，同时制作过程工作量比较大。本附录希望尝试通过自建简单的数据库软件，帮助游戏设计数据文件。

B.1　数据库要求

制作一个简单的数据库主要包括两个问题：

（1）数据在内存中处理的时候所使用的结构；

（2）数据在文件和内存间的读取问题。

这里设计的简单数据库程序将对字段的数目、字段长度和字段名称进行固定，大家可以通过对它的结构进行改进来实现通用的数据库软件。

本数据库字段包括：

（1）序号；

（2）姓名；

（3）性别；

（4）出生年月；

（5）经验值；

（6）体力值；

（7）敏捷度。

本数据库的功能：

（1）输入记录；

（2）显示记录；

（3）删除记录；

（4）插入记录；

（5）查找记录；

（6）修改记录；

（7）排序记录；

（8）保存数据库；

（9）载入数据库；

（10）清空记录。

通过双向链表将每条记录进行连接，在保存的时候每个记录单元之间固定长度，记录之间通过换行分开。

B.2　详 细 设 计

对于字段以下给出了它使用的变量类型和每个字段相关的限制要求：

（1）序号：为大于 0 的数字；

（2）姓名：小于、等于 10 个字符的字符串；

（3）性别：一个字符，M 表示男、F 表示女；

（4）出生年月：日期型（date），不允许出现不存在的日、月、年；

（5）经验值：0-100 的数字；

（6）体力值：0-100 的数字；

（7）敏捷度：0-100 的数字。

根据以上要求建立如下结构：

```
struct node {
    num_range1 number;//序号
    char name[11];//姓名
    char sex;//性别
    struct date birthdate;//生日
    num_range2 experience;//经验值
    num_range2 force;//体力值
    num_range2 smartness;//敏捷度
    struct node *front;//指向上一个节点
    struct node *next;//指向下一个节点
};
```

对于数据库功能进行的模块化设计：

（1）输入记录模块实现功能：能把用户输入的数据，添加进链表；

（2）错误检查模块实现功能：能检查输入的数据数据类型等问题；

（3）显示记录模块实现功能：能从链表中逐一把数据按一定的格式输出到屏幕；

（4）删除记录模块实现功能：能把用户指定的符合条件的结点从链表中去除，并释放内存空间；

（5）插入记录模块实现功能：能把用户再次输入的数据插在链表的中间某一个结点；

（6）查找记录模块实现功能：能在链表中搜索到用户指定的符合条件的结点，并把数据输出到屏幕；

（7）修改记录模块实现功能：能让用户修改指定结点的数据；

（8）排序记录模块实现功能：能按照指定的关键字对链表进行排序；

（9）交换结点模块实现功能：能使相邻的两个结点交换；

（10）保存数据库模块实现功能：能将链表中的数据保存为文件形式，以长期保存；

（11）载入数据库模块实现功能：能把磁盘上的数据库文件载入链表；

（12）命令解释模块实现功能：能对用户输入的命令进行相应的操作；

（13）清空记录模块实现功能：将所有节点清空。

对于链表指针进行的操作：

（1）初始动态链表；

（2）添加结点；

（3）删除结点；

（4）插入结点；

（5）查找结点；

（6）排序结点；

（7）修改结点；

（8）交换结点；

数据库命令集：

（1）APPEND

（2）DELETE

（3）SAVE

（4）LOAD

（5）INSERT

（6）SHOW

（7）FIND

（8）SORT

（9）MODIFY

（10）QUIT

（11）HELP

（12）CLOSE

搜索、修改和排序记录的条件参数：

（1）记录号；

（2）序号；

（3）姓名；

（4）性别；

（5）出生年月；

（6）经验值；

（7）体力值；

（8）敏捷度。

B.3 模块设计

B.3.1 输入

输入记录模块由循环输入记录函数和加入链表函数三部分组成，在模块中同时调用了错误检查模块函数。

输入函数将键盘输入的记录单元放入对应的全局变量中，然后通过错误检查函数来判断输入的有效性，如果输入有效则将全局变量通过节点增加函数放入新建的记录节点，从而使用户输入数据链入内存中的记录链表。

循环输入记录函数：

```
void input_record (void)
{
int n,err;
n = count + 1;
printf("\n");
printf("after enter\n");
number_c=0;
while ( number_c!=-1 ) {
 printf("record %d\n",n);
 printf("No:");
 scanf("%d",&number_c);//读取序号
 printf("name:");
 gets(name_c); //读取姓名
 gets(name_c);
 printf("sex(M,F):");
 scanf("%c",&sex_c); //读取性别
 printf("birthda_day(1978 16 1):");
 scanf("%d %d %d",&birthdate_c.da_year,&birthdate_c.da_day,&birthdate_c.da_mon);
//读取生日
     printf("experience:");
     scanf("%d",&experience_c); //读取经验值
     printf("force:");
     scanf("%d",&force_c); //读取体力值
     printf("smartness:");
     scanf("%d",&smartness_c); //读取敏捷度
     printf("\n");
     err = check_record(&number_c,&sex_c,&birthdate_c,&experience_c,
          &force_c,&smartness_c);//调用数据合法性检测函数
```

```
switch (err) {//错误判断
    case 0://输入无错
     add_record();//将输入数据调入链表函数
     n = n + 1;
     break;
    case 1://序号出错
     printf("no wrong:1-1000\n");
     printf("try again\n");
     break;
    case 2:
     printf("sex wrong:M male F female\n");
     printf("try again\n");
     break;
    case 3:
     printf("birthday wrong:1978 1 16\n");
     printf("try again\n");
     break;
    case 4:
     printf("experience wrong:0-100\n");
     printf("try again\n");
     break;
    case 5:
     printf("force wrong:0-100\n");
     printf("try again\n");
     break;
    case 6:
         printf("smartness wrong:0-100\n");
         printf("try again\n");
         break;
     }
     printf("enter to continue,-1 to end");
     printf("\n");
    }
}
```

将键盘输入的新记录加入链表的函数主要完成申请新的节点空间，将上一个节点的next链表指针指向新申请节点地址，而将新申请节点的front链表指针指向上一个节点地址。同时将数据放入节点。此处还有一个判断是否是头节点的任务，如果是头节点则其 front链表指针指向空。

加入链表函数：

```
void add_record (void)
{
struct node *tmp;
    if (head == nil) {//链表中没有记录的情况
     tmp = (stu_p) malloc (sizeof(struct node));//为当前数据申请空间
     //链表设置
     head = tmp;
     head->next = nil;
     current = head;
     current->front = nil;
     last = current->next;
    }
    else  {
     tmp = (stu_p) malloc (sizeof(struct node));
        tmp->next = nil;
     current->next = tmp;
        tmp->front = current;
     current = tmp;
        last = current->next;
}
//数据赋值
    current->number = number_c;
        strcpy (current->name, name_c);
        current->sex = sex_c;
        current->birthdate = birthdate_c;
        current->experience = experience_c;
        current->force = force_c;
        current->smartness = smartness_c;
        count = count + 1;

    }
```

B.3.2　检查

错误检查模块由记录检查函数和若干记录单元检查函数组成。

错误检查函数除了需要检查输入数据的类型正确性，同时更主要是检查输入数据的逻辑正确性。它通过调用每个字段的检查函数来实现整个记录有效性的检查。

记录检查函数：

```
error_num check record (num rangel *number_c, char *sex_c, struct date
*birthdate_c, num_range2 *experience_c, num_range2 *force c, num_range2 *smartness_c)
{
int err_p;
error_num function;

err_p = FALSE;
//检查序号
    if ((err_p == FALSE))
     if (check_number(*number_c))
         err_p = FALSE;
     else  {
         err_p = TRUE;
         function = 1;
     }
//检查性别
    if ((err_p == FALSE))
     if (check_sex(*sex_c))
         err_p = FALSE;
     else  {
            err_p = TRUE;
            function = 2;
        }
//检查生日
    if ((err_p == FALSE))
        if (check_birthdate(*birthdate_c))
            err_p = FALSE;
     else  {
            err_p = TRUE;
            function = 3;
     }
    //检查经验值
    if ((err_p == FALSE))
```

```
    if (check_experience(*experience_c))
        err_p - FALSE;
    else  {
        err_p - TRUE;
     function - 4;
    }
 //检查体力值
if ((err_p == FALSE))
 if (check_force(*force_c))
     err_p = FALSE;
    else  {
     err_p = TRUE;
        function = 5;
    }
 //检查敏捷度
if ((err_p == FALSE))
    if (check_smartness(*smartness_c))
        err_p = FALSE;
 else  {
        err_p = TRUE;
     function = 6;
 }
 //返回错误代码
if (err_p == FALSE)
 function = 0;

return(function);
}
```

对于游戏对像序号，规定其必须视 1 到 1000 之间的数字。

序号检查函数：

```
int  check_number (num_rangel check_object)
{
int function;
//序号不在 1 到 1000 中的检测
    if ((check_object >= 1) && (check_object <= 1000))
     function = TRUE;
    else
```

```
    function = FALSE;

    return(function);
}
```

对于性别标志的检查，主要判断其是否在“m”“M”“f”“F”四个字符中。

性别检查函数：

```
int  check_sex (char check_object)
{
int function;
//性别输入字符范围检测
    if (check_object=='m'||check_object=='M'||check_object=='f'||check_object=='M')
     function = TRUE;
    else
     function = FALSE;

     return(function);
}
```

对于生日要判别日期的真实性，年必须在公元 1000 到 3000 年之间。同时要判断每个月所有的日期，比如 1.3.5.7.8.10.12 月 31 天一个月，4.6.9.11 月 30 天一个月，而 2 月只有 29 天，如果是 2 月还要判断是否是润年，如果湿润年则只有 28 天。

生日检查函数：

```
int  check_birthdate (struct date check_object)
{
int function;
//检测年是否在 1000-3000 种
if ((check_object.da_year >= 1000) && (check_object.da_year <= 3000))
//检测出生日在月中有 31 号的情况
     if (check_object.da_mon==1||check_object.da_mon==3||
     check_object.da_mon==5||check_object.da_mon==7||check_object.da_mon==8||
     check_object.da_mon==10||check_object.da_mon==12)
          if (check_object.da_day<=31&&check_object.da_day>=1)
           function = TRUE;
          else
           function = FALSE;
//检测出生日在月中没有 31 号的情况
     else if (check_object.da_mon==4||check_object.da_mon==6||
```

```
        check_object.da_mon==9||check_object.da_mon==11)
            if (check_object.da_day<=30&&check_object.da_day>=1)
             function = TRUE;
            else
             function = FALSE;
    //检测出生日在润年2月的情况
        else if (check_object.da_mon == 2)
            if ((check_object.da_year % 4 == 0) &&
                 (check_object.da_year % 100 != 0) ||
                 (check_object.da_year % 400 == 0))
             if (check_object.da_day<=29&&check_object.da_day>=1)
                 function = TRUE;
             else
                 function = FALSE.
            else if (check_object.da_day<=28&&check_object.da_day>=1)
             function = TRUE.
            else
             function = FALSE;
         else
            function = FALSE;
        else
         function = FALSE;

        return(function);
}
```

经验值、体力值和敏捷度都要求是1到100之间的数字。

经验值检查函数：

```
int  check_experience (num_range2 check_object)
{
int function;
//检测经验值是否在1-100
    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
```

```
}
```

体力值检查函数：

```
int  check_force (num_range2 check_object)
{
int function;
//检测体力值是否在 1-100
    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
}
```

敏捷度检查函数：

```
int  check_smartness (num_range2 check_object)
{

int function;

//检测敏捷读是否在 1-100
    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
}
```

B.3.3　显示

记录显示模块由循环显示记录函数、显示节点函数和显示字段名称函数组成。

显示记录函数就是对链表从头节点开始进行遍历，在每次调用节点显示函数后将当前节点设置为当前节点的下一个节点，直到最后一个节点。

循环显示记录函数：

```
void output_record (void)
{
```

```
int n,
    i;
struct node *c,
            *l;

    if (head == nil)
     printf("no record\n");
else  {
//指针初始化
        n = 1;
        c = head;
        l = head->next;
    //显示字段名称
        output_title();
//显示所有链表中的纪录
        for (i = 1; i <= count; i++) {
            output_it(c,n);//显示当前节点纪录
            n = n + 1;
            c = l;
         l = c->next;
        }
    }

}
```

显示节点函数：

```
void output_it (struct node *c,int n)
{
//数据输出
    printf("%7d",n);
    printf("%7d",c->number);
    printf("%10s",c->name);
    printf("%6c",c->sex);
    printf("%6d/%2d/%2d",c->birthdate.da_year,c->birthdate.da_mon,
        c->birthdate.da_day);
    printf("%10d",c->experience);
    printf("%10d",c->force);
    printf("%10d",c->smartness);
```

```
    printf("\n");

}
```

显示字段名称函数：

```
void output_title(void)
{
//显示字段名称
    printf("\n");
    printf("record no name  sex  birthday  experience  force  smartness\n");
}
```

B.3.4 删除

记录删除模块由记录删除条件函数、条件输入函数和删除节点函数组成。

记录删除条件函数实际就是询问用户根据哪个字段进行匹配，同时进行遍历和相应的匹配查找，找到了就调用节点删除函数。

记录删除条件函数：

```
void del_record(void)
{
int condition_code;
int i,
    deleted_num;

int long_c,long_d;
int del_num;
num_rangel del_number;
char del_name[11];
char del_sex;
struct date del_birthdate;
num_range2 del_experience;
num_range2 del_force;
num_range2 del_smartness;
struct node *c_bak,
            *l_bak;
//删除条件
    condition_code = condition_select("del condition:");
    c_bak = current;
    l_bak = last;
```

```
switch (condition_code) {//根据删除条件对应删除链表中的节点记录
 case 1:
     printf("\n");
     printf("record:");
     scanf("%d",&del_num);//读取输入的要删除的记录号
     if ((del_num >= 1) && (del_num <= count)) {//删除记录号在链表记录号中
      if (head != nil) {
                current = nil;
          last = nil;
                for (i = 1; i <= del_num; i++) {
           if (i == 1) {
               current = head;
               last = current->next;
           }
           else  {
               current = last;
               last = current->next;
           }
          }
          delete_it(&current,&last);//删除记录
          printf(" record %d delete\n",del_num);
      }
      else
          printf("no record like u need\n");
     }
     else
      printf("out record\n");
     break;
 case 2:
     printf("\n");
     printf("no:");
     scanf("%d",&del_number);
     if ((del_number >= 1) && (del_number <= 1000)) {//检测要删除序号是否存在
      if (head != nil) {
                current = head;
                last = current->next;
          deleted_num = 0;
                for (i = 1; i <= count; i++) {
```

```
 if (current->number == del_number) {
                delete_it(&current,&last);
     deleted_num = deleted_num + 1;
 }
           else  {
     current = last;
     last = current->next;
 }
}
if (deleted_num != 0)
 printf(" record %d delete\n",deleted_num);
else
     printf("no record like u need\n");
  }
  else
     printf("no record like u need\n");
 }
 else
  printf("out record\n");
 break;
case 3:
 printf("\n");
 printf("name:");
 gets(del_name);
 if (head != nil) {
  current = head;
  last = current->next;
  deleted_num = 0;
for (i = 1; i <= count; i++) {
 if (strcmp(current->name,del_name) == 0) {//检测要删除姓名是否存在
  delete_it(&current,&last);
  deleted_num = deleted_num + 1;
 }
 else  {
  current = last;
  last = current->next;
 }
}
```

```
        if (deleted_num != 0)
            printf(" record %d delete\n",deleted_num);
        else
            printf("no record like u need\n");
      }
      else
       printf("out rccord\n");
      break;
case 4:
        printf("\n");
      printf("sex:");
      scanf("%c",&del_sex);
      if (head != nil) {
      current = head;
      last = current->next;
      deleted_num = 0;
            for (i = 1; i <= count; i++) {
                if ((current->sex == del_sex)) {
                    delete_it(&current,&last);
                    deleted_num = deleted_num + 1;
                }
          else  {
           current = last;
           last = current->next;
          }
      }
      if (deleted_num != 0)
          printf(" record %d delete\n",deleted_num);
      else
           printf("no record like u need\n");
          }
          else
           printf("out record\n");
          break;
      case 5:
          printf("\n");
          printf("birthday:");
          scanf("%d%d%d",&del_birthdate.da_year,&del_birthdate.da_mon,
```

```
        &del_birthdate.da_day);
    if (head != nil) {
     current = head;
     last = current->next;
     deleted_num = 0;
     long_d = del_birthdate.da_year;
     long_d = long_d * 100;
     long_d = long_d + del_birthdate.da_mon;
     long_d = long_d * 100;
     long_d = long_d + del_birthdate.da_day;
     for (i = 1; i <= count; i++) {
         long_c = current->birthdate.da_year;
         long_c = long_c * 100;
         long_c = long_c + current->birthdate.da_mon;
         long_c = long_c * 100;
         long_c = long_c + current->birthdate.da_day;
         if ((long_c == long_d)) {
          delete_it(&current,&last);
          deleted_num = deleted_num + 1;
         }
         else  {
          current = last;
          last = current->next;
         }
    }
    if (deleted_num != 0)
        printf(" record %d delete\n",deleted_num);
    else
        printf("no record like u need\n");
    }
    else
     printf("out record\n");
     break;
    case 6:
       printf("\n");
     printf("experience:");
     scanf("%d",&del_experience);
     if (head != nil) {
```

```
current = head;
last = current->next;
deleted_num = 0;
for (i = 1; i <= count; i++) {
    if ((current->experience == del_experience)) {
     delete_it(&current,&last);
     deleted_num = deleted_num + 1;
          }
    else  {
     current = last;
     last = current->next;
    }
    }
    if (deleted_num != 0)
        printf(" record %d delete\n",deleted_num);
    else
        printf("no record like u need\n");
    }
    else
    printf("out record\n");
    break;
case 7:
    printf("\n");
    printf("force:");
    scanf("%d",&del_force);
       if (head != nil) {
     current = head;
       last = current->next;
deleted_num = 0;
       for (i = 1; i <= count; i++) {
           if ((current->force == del_force)) {
               delete_it(&current,&last);
     deleted_num = deleted_num + 1;
    }
    else  {
               current = last;
     last = current->next;
    }
```

```
        }
        if (deleted_num != 0)
            printf(" record %d delete\n",deleted_num);
        else
            printf("no record like u need\n");
       }
       else
        printf("out record\n");
       break;
case 8:
    printf("\n");
    printf("smartness:");
    scanf("%d",&del_smartness);
      if (head != nil) {
    current = head;
          last = current->next;
    deleted_num = 0;
          for (i = 1; i <= count; i++) {
              if ((current->smartness == del_smartness)) {
        delete_it(&current,&last);
                  deleted_num = deleted_num + 1;
       }
       else  {
                  current = last;
        last = current->next;
       }
}
if (deleted_num != 0)
            printf(" record %d delete\n",deleted_num);
        else
            printf("no record like u need\n");
       }
       else
        printf("out record\n");
       break;
default:
    printf("delete error\n");
    break;
```

```
    }
    if (last != nil) {
     current = c_bak;
     last = l_bak;
    }

}
```

条件输入函数：

```
int  condition_select (char *condition_message)
{
int condition_code;
int function;
//显示并且读取选择的条件
    printf("\n");
    printf("%s\n",condition_message);
    printf("          1 record 2 No 3 name 4 sex\n");

    printf("          5 birthday 6 experience 7 force 8 smartness\n");

    printf("\n");
    printf("choise:(1~8)");
    scanf("%d",&condition_code);
    function = condition_code;

    return(function);
}
```

删除节点函数需要将当前节点的上一个节点的 next 指针指向当前节点下一个节点的地址，同时将当前节点下一个节点的 front 指针指向当前节点上一个节点的地址，最后释放当前节点空间。这里需要考虑 4 种情况：

（1）只有一个节点，删除后就没有节点了；

（2）要删除的节点是第一个节点，删除后其下一个节点为头节点；

（3）要删除的节点是最后一个节点，删除后其上一个节点为最后一个节点；

（4）其他情况。

删除节点函数：

```
void delete_it (struct node **current,struct node  **last)
{
```

```
//可以自动后移指针
struct node *tmp;

    if ((*current == head) && (*last == nil)) {     //只有一个节点
        free(*current);
        head = nil;
     *current = nil;
        *last = nil;
        count = 0;
    }
    else if ((*current != head) && (*last == nil)) {//删除尾节点
        *current = (*current)->front;
     free(*current);
        (*current)->next = *last;
        count = count - 1;
    }
    else if ((*current == head) && (*last != nil)) {//删除头节点
        head = *last;
     free(*current);
        *current = head;
        (*current)->front = nil;
     *last = (*current)->next;
        count = count - 1;
    }
    else  {//其他情况
        (*current)->front->next = (*current)->next;
        (*current)->next->front = (*current)->front;
        tmp = *current;
        *current = *last;
        *last = (*current)->next;
     free(tmp);
        tmp = nil;
        count = count - 1;
    }
}
```

B.3.5　插入

记录插入模块由记录插入条件函数、节点插入函数和加入链表函数组成。

记录插入条件函数要求用户输入插入记录的号码，然后根据插入记录号码找到小于等

于该记录的节点，在它后面调用节点插入函数。

记录插入条件函数：

```
void insert_record (void)
{
int i,
    n,
    p;
struct node *c,
        *l;

    if (head != nil) {
        printf("\n");
printf("insert record:");
scanf("%d",&n);//确定插入位置
if ((n >= 1) && (n <= count)) {
    c = nil;
       l = nil;
       for (i = 1; i <= n; i++) {
           if (i == 1) {
               c = head;
               l = c->next;
           }
           else  {
               c = l;
               l = c->next;
           }
       }
    insert_it(&c,n,&p);//插入节点
       if (p == 1)
        printf("insert over\n");
       else
        printf("insert wrong\n");
 }
 else
     printf("out record\n");
}
else  {//没有记录插入第一个节点
 insert_it(&head,1,&p);
```

```
        if (p == 1)
         printf("insert over\n");
        else
         printf("insert wrong\n");
    }

}
```

节点插入函数接收用户输入，对插入的记录进行有有效性检查，然后申请新的节点空间键数据放入，同时将前面找到的节点的 next 指针复制给新节点的 next 指针，并将那个 next 指针指向新节点，将新节点 next 指针指向的节点的 front 指针指向新节点，将新节点的 front 指针指向前面找到的节点。

节点插入函数：

```
void insert_it (struct node **c, int n, int *p)
{
struct node *insert;
error_num err;
//接收数据输入
    *p = 0;
     printf("record %d\n", n);
     printf("No:");
     scanf("%d", &number_c);
     printf("name:");
     gets(name_c);
     gets(name_c);
     printf("sex(M,F):");
     scanf("%c", &sex_c);
     printf("birthda_day(1978 16 1):");
     scanf("%d %d %d", &birthdate_c.da_year, &birthdate_c.da_day, &birthdate_c.da_mon);
     printf("experience:");
     scanf("%d", &experience_c);
     printf("force:");
     scanf("%d", &force_c);
     printf("smartness:");
     scanf("%d", &smartness_c);
     printf("\n");
     err = check_record(&number_c, &sex_c, &birthdate_c, &experience_c,
          &force_c, &smartness_c);//错误检测
```

```
switch (err) {
    case 0:
     add_record();
     n = n + 1;
     *p=1;
     break;
    case 1:
     printf("no wrong:1-1000\n");
     printf("try again\n");
     break;
    case 2:
     printf("sex wrong:M male F female\n");
     printf("try again\n");
     break;
    case 3:
     printf("birthday wrong:1978 1 16\n");
     printf("try again\n");
     break;
    case 4:
     printf("experience wrong:0-100\n");
     printf("try again\n");
     break;
    case 5:
     printf("force wrong:0-100\n");
     printf("try again\n");
     break;
    case 6:
     printf("smartness wrong:0-100\n");
     printf("try again\n");
     break;
}
if (*p == 1) {
    if (*c == nil) {
        add_record();//增加节点
```

```
}    //插入于第一个节点
else if (*c == head) {
        insert = (stu_p) malloc (sizeof(struct node));
        insert->front = nil;
        insert->next = nil;
    head->front = insert;
        insert->next = head;
    head = insert;
        insert->number = number_c;
        strcpy (insert->name,name_c);
        insert->sex = sex_c;
        insert->birthdate = birthdate_c;
        insert->experience = experience_c;
 insert->force = force_c;
        insert->smartness = smartness_c;
        count = count + 1;
}
else  {//不是第一个节点
        insert = (stu_p) malloc (sizeof(struct node));
        insert->front = nil;
 insert->next = nil;
        (*c)->front->next = insert;
        insert->front = (*c)->front;
        (*c)->front = insert;
        insert->next = *c;
 insert->number = number_c;
        strcpy (insert->name,name_c);
 insert->sex = sex_c;
        insert->birthdate = birthdate_c;
        insert->experience = experience_c;
        insert->force = force_c;
        insert->smartness = smartness_c;
        count = count + 1;
    }
   }

}
```

B.3.6　查找

记录查找模块在查找到节点后同时调用了显示节点函数和显示字段名称函数。

记录查找函数根据用户输入查找条件和查找关键数据，在链表中进行遍历，同时进行对应的匹配查询，如果相等就显示对应节点。

记录查找函数：

```
void find_record (void)
{
int condition_code;
int i,
    found_num;

int long_c,long_f;
struct node *c,
           *l;
int find_num;
num_rangel find_number;
char find_name[11];
char find_sex;
struct date find_birthdate;
num_range2 find_experience;
num_range2 find_force;
num_range2 find_smartness;

    condition_code = condition_select("find condition:");//查找条件
    switch (condition_code) {
     case 1://按照记录号查找
         printf("\n");
         printf("record:");
         scanf("%d",&find_num);//读取查找的记录号
         if ((find_num >= 1) && (find_num <= count)) {
               if (head != nil) {
                   c = nil;
                   l = nil;
                   found_num = 1;
             for (i = 1; i <= find_num; i++) {
                       if (i == 1) {
                           c = head;
                           l = c->next;
```

```
                }
                else  {
                    c = l;
                    l = c->next;
                ;
            }
            output_title();
        output_it(c,find_num);//输出查找到的节点
        printf("find %d\n",found_num);
    }
    else
        printf("no record\n");
}
    else
 printf("out record\n");
 break;
case 2:
    printf("\n");
 printf("no:");
 scanf("%d",&find_number);
 if (head != nil) {
        c = head;
        l = c->next;
        found_num = 0;
        output_title();
        for (i = 1; i <= count; i++) {
            if (c->number == find_number) {
      output_it(c,i);
                found_num = found_num + 1;
    c = l;
                c = c->next;
    }
    else  {
                c = l;
                l = c->next;
    }
}
if (found_num != 0)
```

```
            printf("find %d\n", found_num);
        else
            printf("no record\n");
        }
        else
        printf("out record\n");
        break;
    case 3:
        printf("\n");
        printf("name:");
        gets(find_name);
        if (head != nil) {
    c = head;
    l = c->next;
          found_num = 0;
          output_title();
          for (i = 1; i <= count; i++) {
              if (strcmp(c->name, find_name) == 0) {
        output_it(c, i);
                  found_num = found_num + 1;
        c = l;
        c = c->next;
        }
        else {
        c = l;
        l = c->next;
        }
        if (found_num != 0)
            printf("find %d\n", found_num);
        else
            printf("no record\n");
        }
        else
        printf("out record\n");
        break;
    case 4:
        printf("\n");
```

```
printf("sex:");
scanf("%c",&find_sex);
   if (head != nil) {
       c = head;
l = c->next;
       found_num = 0;
       output_title();
       for (i = 1; i <= count; i++) {
           if (c->sex == find_sex) {
     output_it(c,i);
                found_num = found_num + 1;
     c = l;
                c = c->next;
          }
          else {
           c = l;
           l = c->next;
          }
      }
      if (found_num != 0)
          printf("find %d\n",found_num);
      else
          printf("no record\n");
     }
     else
      printf("out record\n");
     break;
case 5:
     printf("\n");
     printf("birthday:");
     scanf("%d%d%d",&find_birthdate.da_year,&find_birthdate.da_mon,
          &find_birthdate.da_day);
     if (head != nil) {
      c = head;
      l = c->next;
      found_num = 0;
           output_title();
      long_f = find_birthdate.da_year;
```

```
            long_f = long_f * 100;
        long_f = long_f + find_birthdate.da_mon;
        long_f = long_f * 100;
        long_f = long_f + find_birthdate.da_day;
        for (i = 1; i <= count; i++) {
            long_c = c->birthdate.da_year;
                long_c = long_c * 100;
            long_c = long_c + c->birthdate.da_mon;
                long_c = long_c * 100;
            long_c = long_c + c->birthdate.da_day;
                if (long_c == long_f) {
             output_it(c, i);
                    found_num = found_num + 1;
                    c = 1;
             c = c->next;
            }
                else {
             c = 1;
                    1 = c->next;
                }
           }
         if (found_num != 0)
            printf("find %d\n", found_num);
         else
            printf("no record\n");
        }
        else
         printf("out record\n");
        break;
        case 6:
            printf("\n");
         printf("experience:");
         scanf("%d", &find_experience);
            if (head != nil) {
                c = head;
                1 = c->next;
                found_num = 0;
         output_title();
```

```
for (i - 1; i <= count; i++) {
            if (c >experience == find_experience) {
                output_it(c,i);
                found num = found_num + 1,
     c = 1;
     l = c->next;
    }
    else  {
     c = l;
     l = c->next,
    }
}
if (found_num !- 0)
     printf("find %d\n",found_num);
     else
     printf("no record\n");
    }
    else
     printf("out record\n");
    break;
case 7:
    printf("\n");
    printf("force:");
    scanf("%d",&find_force);
    if (head != nil) {
     c = head;
     l = c->next;
     found_num = 0;
output_title();
for (i = 1; i <= count; i++) {
    if (c->force == find_force)
     output_it(c,i);
     found_num = found_num + 1;
     c = l;
     l = c->next;
    }
    else  {
     c = l;
```

```
                1 = c->next;
               }
          }
          if (found_num != 0)
               printf("find %d\n", found_num);
          else
               printf("no record\n");
          }
          else
          printf("out record\n");
          break;
     case 8:
          printf("\n");
          printf("\n");
          printf("smartness:");
          scanf("%d",&find_smartness);
          if (head != nil) {
          c = head;
          l = c->next;
                found_num = 0;
                output_title();
                for (i = 1; i <= count; i++) {
               if (c->smartness == find_smartness) {
               output_it(c, i);
                         found_num = found_num + 1;
               c = l;
               l = c->next;
               }
               else  {
               c = l;
               l = c->next;
               }
          }
          if (found_num != 0)
               printf("find %d\n", found_num);
          else
               printf("no record\n");
     }
          else
```

```
            printf("out record\n");
            break;
        default:
            printf("find error\n");
            break;
        }

}
```

B.3.7　修改

记录修改模块包括记录修改条件函数和节点修改函数。

记录修改函数根据用户输入条件和关键数据在链表中遍历并且进行对应的匹配查找，找到后调用节点修改函数对节点进行修改。

记录修改条件函数：

```
void modify_record (void)
{
int condition_code;
int i,
    modified_num;


int long_c,long_m;
struct node *c,
        *l;
int mdf_num;
num_range1 mdf_number;
char mdf_name[11];
char mdf_sex;
struct date mdf_birthdate;
num_range2 mdf_experience;
num_range2 mdf_force;
num_range2 mdf_smartness;

    condition_code = condition_select("modify:");//修改条件
    switch (condition_code) {
     case 1://按记录号修改
         printf("\n");
```

```
printf("record:");
scanf("%d",&mdf_num);//读取记录号
   if ((mdf num >= 1) && (mdf_num <= count)) {
if (head != nil) {
          c = nil;
          l = nil,
          for (i - 1; i <= mdf num; i++) {
    if (i -- 1) {
                  c - head;
                  l - c->next;
    }
          else {
                  c = l;
              l = c->next;
         }
              }
              modify it(c,l,i);//修改节点
        printf("%d modify\n",mdf_num);
     }
     else
     '   printf("no record\n");
    }
       else
        printf("out record\n");
    break;
case 2:
    printf("\n");
    printf("no:");
    scanf("%d",&mdf_number);
       if (head != nil) {
    c = head;
    l = c->next;
    modified_num = 0;
    for (i = 1; i <= count; i-+) {
        if (c->number == mdf_number) {
         modify_it(c,l,i);
         modified_num = modified_num + 1;
         c = l;
```

```
        c = c->next;
      }
      else
        c = l;
        l = c->next;
      }
  }
  if (modified_num != 0)
      printf(" %d modify\n",modified_num);
  else
      printf("no record\n");
}
else
      printf("no record\n");
  break;
case 3:
  printf("\n");
  printf("name:");
  gets(mdf_name);
  if (head != nil) {
   c = head;
   l = c->next;
   modified_num = 0;
   for (i = 1; i <= count; i++) {
            if (strcmp(c->name,mdf_name) == 0) {
                modify_it(c,l,i);
       modified_num = modified_num + 1;
                c = l;
       c = c->next;
      }
      else {
       c = l;
       l = c->next;
      }
  }
  if (modified_num != 0)
      printf(" %d modify\n",modified_num);
  else
```

```
            printf("no record\n");
        }
        else
            printf("no record\n");
        break;
    case 4:
        printf("\n");
        printf("sex:");
        scanf("%c",&mdf_sex);
        if (head != nil) {
         c = head;
         l = c->next;
         modified_num = 0;
    for (i = 1; i <= count; i++) {
        if (c->sex == mdf_sex) {
         modify_it(c,l,i);
         modified_num = modified_num + 1;
                    c = l.
         c = c->next;
               }
               else  {
                   c = l;
                   l = c->next;
        }
    }
    if (modified_num != 0)
        printf(" %d modify\n",modified_num);
    else
            printf("no record\n");
        }
        else
            printf("no record\n");
        break;
    case 5:
        printf("\n").
        printf("birthday:");
        scanf("%d%d%d",&mdf_birthdate.da_year,&mdf_birthdate.da_mon,
            &mdf_birthdate.da_day);
```

```
if (head != nil) :
c = head;
        l = c->next,
        modified_num = 0;
long_m = mdf_birthdate.da_year;
        long_m = long_m * 100;
long_m = long_m + mdf_birthdate.da_mon;
        long_m = long_m * 100;
long_m = long_m + mdf_birthdate.da_day;
for (i = 1; i <= count; i++) {
    long_c = c->birthdate.da_year;
            long_c = long_c * 100;
        long_c = long_c + c->birthdate.da_mon;
        long_c = long_c * 100;
        long_c = long_c + c->birthdate.da_day;
        if ((long_c == long_m)) {
                modify_it(c, l, i);
         modified_num = modified_num + 1;
                c = l;
                c = c->next;
            }
            else  {
                c = l;
                l = c->next;
            }
}
    if (modified_num != 0)
    printf(" %d modify\n",modified_num);
     else
        printf("no record\n");
    }
    else
        printf("no record\n");
    break;
case 6:
    printf("\n");
    printf("experience:");
    scanf("%d",&mdf_experience);
```

```
if (head != nil) {
    c = head;
    l = c->next;
    modified num - 0;
    for (i - 1; i <- count; i++) {
        if (c->experience == mdf_experience) {
 modify it(c, l, i);
 modified_num = modified_num + l;
 c = l;
 c = c->next;
 }
 else {
     c = l;
     l = c->next;
     }
 }
 if (modified num !- 0)
     printf(" %d modify\n", modified_num);
 else
     printf("no record\n");
 }
 else
     printf("no record\n");
 break;
case 7:
 printf("\n");
 printf("force:");
 scanf("%d", &mdf_force);
 if (head != nil) {
 c = head;
 l = c->next;
 modified_num = 0;
 for (i = 1; i <= count; i++) {
     if (c->force == mdf_force) {
     modify_it(c, 1, i);
     modified_num = modified_num + 1;
     c = l;
     c = c->next;
```

```
        }
        else {
         c = l;
         l = c->next;
        }
    }
    if (modified_num != 0)
        printf(" %d modify\n",modified_num);
    else
        printf("no record\n");
}
else
    printf("no record\n");
    break;
case 8:
    printf("\n");
    printf("smartness:");
    scanf("%d",&mdf_smartness);
       if (head != nil) {
    c = head;
          l = c->next;
    modified_num = 0;
          for (i = 1; i <= count; i++) {
              if (c->smartness == mdf_smartness) {
                  modify_it(c,l,i);
                  modified_num = modified_num + 1;
                  c = l;
        c = c->next;
              }
              else {
                  c = l;
                  l = c->next;
    }
}
if (modified_num != 0)
    printf(" %d modify\n",modified_num);
else
    printf("no record\n");
```

```
        }
        else
            printf("no record\n");
        break;
    default:
        printf("modify error\n");
        break;
    }

}
```

节点修改函数根据获得的节点，将用户新输入的数据在有效性检查后放入节点。如果碰到输入-1 表示对应的单元数据不进行修改。

节点修改函数：

```
void modify_it (struct node *c,struct node *l,int n)
{
//读取节点新数据
    printf("modify %d input -1 ,still\n",n);
    printf("record %d\n",n);
    printf("No:");
    scanf("%d",&number_c);
    printf("name:");
    gets(name_c);
    gets(name_c);
    printf("sex(M,F):");
    scanf("%c",&sex_c);
    printf("birthda_day(1978 16 1):");
    scanf("%d %d %d",&birthdate_c.da_year,&birthdate_c.da_day,&birthdate_c.da_mon);
    printf("experience:");
    scanf("%d",&experience_c);
    printf("force:");
    scanf("%d",&force_c);
    printf("smartness:");
    scanf("%d",&smartness_c);
    printf("\n");
//修改节点，读入-1 不修改
    if (number_c != - 1)
    if (check_number(number_c))
```

```
        c->number = number_c;
    else
        printf("out record\n");
    if (stremp(name_c,"-1") != 0)
    strcpy (c->name, name_c);
    if (sex_c != '-')
    if (check_sex(sex_c))
        c->sex = sex_c;
    else
        printf("sex error\n");
    if ((birthdate_c.da_year != -1) && (birthdate_c.da_mon != -1) &&
        (birthdate_c.da_day != -1))
    if (check_birthdate(birthdate_c))
        c->birthdate = birthdate_c;
    else
        printf("birthday error\n");
    if (experience_c != -1)
    if (check_experience(experience_c))
        c->experience = experience_c;
    else
        printf("experience error\n");
    if (force_c != -1)
    if (check_force(force_c))
        c->force = force_c;
    else
        printf("force error\n");
    if (smartness_c != -1)
    if (check_smartness(smartness_c))
        c->smartness = smartness_c;
    else
        printf("smartness error\n");

}
```

B.3.8 排序和交换节点

排序模块是由记录排序条件函数和交换节点模块组成的。

记录排序条件函数根据用户条件输入通过交换节点来实现沉淀法排序。所谓沉淀法就是在每次遍历和交换后将链表中对应单元最小的数据当到最前面，然后再进行除去前面已

经完成沉淀节点的遍历和交换。

记录排序条件函数：

```
void sort_record (void)
{
int condition_code;
int i,
    j;

int long_c, long f;
struct node *c_bak,
        *l_bak;

condition_code = condition_select("sort condition:");
switch (condition code) {
case 1:
    printf("record over\n");
    break;
case 2:
    if ((head != nil) && (head->next != nil)) {
     c_bak = current;
     l_bak = last;    //沉淀法排序
     for (i = count - 1; i >= 1; i--) {
         current = c_bak;
         last = l_bak;
         for (j = 1; j <= i; j++) {
          if (current->number < current->front->number)
              swap_it(&current, &last, &c_bak, &l_bak);
                  current = current->front;
       last = current->next;
            }
        }
        current = c_bak;
        last = l_bak;
}
printf("no over\n");
break;
case 3:
    if ((head != nil) && (head->next != nil)) {
```

```
    c_bak = current,
    l_bak = last;
    for (i = count - 1; i >= 1; i--) {
        current = c_bak;
        last - l_bak;
        for (j = 1; j <= i; j--) {
            if (strcmp(current->name,
      current->front->name) < 0)
                swap_it(&current,&last,&c_bak,&l_bak);
            current = current->front;
last = current->next;
        }
        current - c_bak;
        last = l_bak;
    }
 printf("name over\n");
    break;
case 4:
 if ((head != nil) && (head->next != nil)) {
        c_bak = current;
        l_bak = last;
        for (i = count - 1; i >= 1; i--) {
            current = c_bak;
            last = l_bak;
            for (j - 1; j <= i; j++) {
                if (current->sex < current->front->sex)
                    swap_it(&current,&last,&c_bak,&l_bak);
                current = current->front;
                last = current->next;
        }
 }
        current = c_bak;
        last = l_bak;
    }
 printf("sex over\n");
 break;
case 5:
```

```
if ((head != nil) && (head->next != nil)) {
    c_bak = current;
    l_bak = last;
    for (i = count - 1; i >= 1; i--) {
        current = c_bak;
 last = l_bak;
        for (j = 1, j <= i; j++) {
 long_c = current->birthdate.da_year;
            long_c = long_c * 100;
 long_c = long_c + current->birthdate.da_mon;
            long_c = long_c * 100;
 long_c = long_c + current->birthdate.da_day;
     long_f = current->front->birthdate.da_year;
            long_f = long_f * 100;
            long_f = long_f +
         current->front->birthdate.da_mon;
            long_f = long_f * 100;
     long_f = long_f + current->front->birthdate.da_day;

     if (long_c < long_f)
         swap_it(&current, &last, &c_bak, &l_bak);
     current = current->front;
     last = current->next;
    }
}
current = c_bak;
last = l_bak;
}
printf("birthday over\n");
break;
case 6:
    if ((head != nil) && (head->next != nil)) {
 c_bak = current;
        l_bak = last;
        for (i = count - 1; i >= 1; i--) {
            current = c_bak;
            last = l_bak;
            for (j = 1; j <= i; j++) {
```

```
                if (current->experience < current->front->experience)
                    swap_it(&current,&last,&c_bak,&l_bak);
                current = current->front;
        last = current->next;
             }
 }
        current = c_bak;
        last = l_bak;
    }
 printf("experience over\n");
 break;
case 7:
    if ((head != nil) && (head->next != nil)) {
        c_bak = current;
 l_bak = last;
 for (i = count - 1; i >= 1; i--) {
            current = c_bak;
            last = l_bak;
            for (j = 1; j <= i; j++) {
                if (current->force < current->front->force)
                    swap_it(&current,&last,&c_bak,&l_bak);
                current = current->front;
                last = current->next;
            }
        }
 current = c_bak;
        last = l_bak;
}
 printf("force over\n");
 break;
case 8:
    if ((head != nil) && (head->next != nil)) {
        c_bak = current;
        l_bak = last;
 for (i = count - 1; i >= 1; i--) {
            current = c_bak;
            last = l_bak;
            for (j = 1; j <= i; j++) {
```

```
                    if (current->smartness < current->front->smartness)
                swap_it(&current,&last,&c_bak,&l_bak);
                    current = current->front;
                    last = current->next;
                }
            }
            current - c_bak;
        last = l_bak;
        }
    printf("smartness over\n");
    break;
}
```

}

交换节点函数就是将 current 指向的节点和 last 指向的节点进行交换。

交换节点模块函数：

```
void swap_it (struct node **current,struct node **last,struct node **c_bak,struct node
**l_bak)
{//用于排序
struct node *tmp_c,
        *tmp_l;
//使 C_BAK 总是指向链表尾部

    if (((*current)->front->front == nil) && (*last == nil)) {
       *current = (*current)->front;
        head = *c_bak;
        (*current)->front = head;
        (*c_bak)->next = *current;
    (*current)->next = nil;
       *last = nil;
    *c_bak = *current;
    *l_bak = nil;
}
else if (((*current)->front->front != nil) && (*last == nil)) {
    *current = (*current)->front;
    (*current)->front->next = *c_bak;
    (*c_bak)->front = (*current)->front;
```

```
    (*c_bak)->next = *current;
    (*current)->front = *c_bak;
    (*current)->next = nil;
    *last = (*current)->next;
*c_bak = *current,
    *l_bak = *last;
}
else if (((*current)->front->front == nil) && (*last != nil)) {
    head = *current;
    *current = (*current)->front;
    head->next->front = *current;
(*current)->next = head->next;
    head->front = nil;
head->next = *current;
    (*current)->front = head;
*last = (*current)->next;
}
else  {
tmp_c = *current;
    *current = (*current)->front;
    (*current)->front->next = tmp_c;
    tmp_c->next->front = *current;
    tmp_c->front = (*current)->front;
    (*current)->next = tmp_c->next;
    tmp_c->next = *current;
    (*current)->front = tmp_c;
}
```

B.3.9 保存

记录链表保存函数实际上就是通过用户给入的文件名，对联表进行遍历将每个节点的数据根据给定空间写入文件，在每个节点输入完成后换行。同时在文件最开始输入记录节点的数量。

保存记录模块函数：

```
void save_record (void)
{
```

```
int n;
char path[81];
struct node *c,
            *l;


    if (head == nil)
     printf("no record\n");
    else  {
     n = 0;     //文件初始化
     printf("\n");
     printf("write file name ps:pb.TXT:\n"),
     gets(path);
     db_file=fopen(path,"wt");     //指针初始化
     c = head;
     l = head->next;
     fprintf(db_file,"%d\n",count);     //写入最大记录数
     while (c != nil) {     //数据输出
         fprintf(db_file,"%d",c->number);
         fprintf(db_file,"%10s",c->name);
         fprintf(db_file,"%2lc",c->sex);
         fprintf(db_file,"%5d%3d%3d",c->birthdate.da_year,
              c->birthdate.da_day,c->birthdate.da_mon);
         fprintf(db_file,"%4d",c->experience);
         fprintf(db_file,"%4d",c->force);
         fprintf(db_file,"%4d",c->smartness);
         fprintf(db_file,"\n");
            n = n + 1;
            c = l;
            l = c->next;
          }
        fclose(db_file);     //保护代码
        if (n != count)
            printf("sum of record wrong\n");
        printf("have been saved\n");
       }
    }
```

B.3.10 读取

读取记录模块由读取记录文件函数和加入链表函数组成。

读取记录模块根据用户要求读出的文件首先读出记录的数量，然后根据记录的数量将所有记录数据通过加入链表函数放入链表节点，者有些类似于记录输入函数。

读取记录文件函数：

```
void read_record (void)
{
int n,i,len;
char *path;
char panduan;

    count = 0;     //文件初始化
    printf("\n");
    printf("give the name of dbf like pb.txt:\n");
    gets(path);
    if((db_file-fopen(path,"rt"))==NULL)
     printf("this file is not exist");
    else
    {
    fseek(db_file,0,SEEK_SET);
    fscanf(db_file,"%d",&n);     //读取最大记录数
    for (i = 1; i <=n; i++) {     //数据载入缓冲
     fscanf(db_file,"%d",&number_c);
     fscanf(db_file,"%10s",&name_c);
     fseek(db_file,1,SEEK_CUR);
     fscanf(db_file,"%c",&sex_c);
     fscanf(db_file,"%5d",&birthdate_c.da_year);
     fscanf(db_file,"%3d",&birthdate_c.da_day);
     fscanf(db_file,"%3d",&birthdate_c.da_mon);
     fscanf(db_file,"%4d",&experience_c);
     fscanf(db_file,"%4d",&force_c);
     fscanf(db_file,"%4d",&smartness_c);
     add_record();
    }
    printf("file read end\n");
    printf("record number is %d \n",i-1);
    }
}
```

B.3.11 清空

清空节点函数，将整个链表释放，使头节点为空。

清空节点模块函数：

```
void close_database (int n)
{//N为内部使用,信息显示开关
struct node *c,
            *l;
int i;
char pan;

    if (head != nil) {
     c = head;
     l = c->next;
     pan = ' ';
     if (n == 1) {
         printf("\n");
         printf("save database?(Y/N)\n");
         scanf("%c",&pan);
         if (pan == 'Y')
                save_record();
            printf("\n");
        }
        for (i = 1; i <= count; i++)
            delete_it(&c,&l);
        head = nil;
        current = nil;
        last = nil;
        count = 0;
        if (n == 1)
         printf("database close\n");
    }
}
```

B.4 程 序 代 码

程序功能：

用 tc 实现简单的数据库功能。从而为今后游戏数据文件提供数据文件制作、数据文件读取、修改、删除和显示的基础。

程序流程：

（1）初始化数据结构；

（2）判断是否是退出命令，如果是退出命令程序结束；

（3）如果不是则判断具体命令，并且调用对应数据库命令模块；

（4）循环 2、3 步骤直到退出命令出现。

主要函数：

```
void command_select (char command_line[])
//数据库命令读取函数
```

程序要点：

本例程和第 11 章结合可以实现你所喜欢的游戏数据文件结构以及数据文件制作工具。

程序代码(data.c)：

```
#include <io.h>
#include <bios.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <mem.h>
#include <math.h>
#include <string.h>
#include <malloc.h>
#include <time.h>

typedef unsigned short     num_range1;
typedef unsigned short      num_range2;
typedef unsigned short      error_num;

#define TRUE 1
#define FALSE 0
#define nil 0

typedef struct node *stu_p;
struct node {
    num_range1 number;
    char name[11];
```

```
    char sex;
    struct date birthdate;
    num_range2 experience;
    num_range2 force;
    num_range2 smartness;
    struct node *front;
    struct node *next;
} ;


FILE *db_file;

struct node *head,
        *current,
        *last;
int count,
    tmp;

char *command_line;

//字段缓冲变量
num_range1 number_c;
char name_c[11];

num_range2 experience_c;
char sex_c;
num_range2 force_c;
num_range2 smartness_c;
struct date birthdate_c;

//过程和函数
student_database();

int  check_number();
int  check_sex();
int  check_birthdate();
int  check_experience();
int  check_force();
```

```
int  check_smartness();

error_num check_record();
int  condition_select();
void output_title();
void output_it();
void modify_it();
void swap_it();
void delete_it();
void add_record();
void insert_it();
void save_record();
void close_database();
void output_record();
void del_record();
void find_record();
void sort_record();
void modify_record();
void read_record();
void input_record();
void insert_record();
void command_select();
void optimize();

int  check_number (num_range1 check_object)
{
int function;
    if ((check_object >= 1) && (check_object <= 1000))
     function = TRUE;
    else
     function = FALSE;

    return(function);
}

int  check_sex (char check_object)
{
int function;
```

```
    if (check_object=='m'||check_object=='M'||check_object=='f'||check_object=='M')
     function = TRUE;
    else
     function = FALSE;

    return(function);
}

int  check_birthdate (struct date check_object)
{

int function;

if ((check_object.da_year >= 1000) && (check_object.da_year <= 3000))
 if (check_object.da_mon==1||check_object.da_mon==3||
 check_object.da_mon==5||check_object.da_mon==7||check_object.da_mon==8||
 check_object.da_mon==10||check_object.da_mon==12)
     if (check_object.da_day<=31&&check_object.da_day>=1)
      function = TRUE;
     else
      function = FALSE;

 else if (check_object.da_mon==4||check_object.da_mon==6||
 check_object.da_mon==9||check_object.da_mon==11)
     if (check_object.da_day<=30&&check_object.da_day>=1)
      function = TRUE;
     else
      function = FALSE;
 else if (check_object.da_mon == 2)
     if ((check_object.da_year % 4 == 0) &&
          (check_object.da_year % 100 != 0) ||
          (check_object.da_year % 400 == 0))
      if (check_object.da_day<=29&&check_object.da_day>=1)
          function = TRUE;
      else
          function = FALSE;
     else if (check_object.da_day<=28&&check_object.da_day>=1)
      function = TRUE;
```

```
        else
         function = FALSE;
     else
        function = FALSE;
    else
     function = FALSE;

    return(function);
}

int  check_experience (num_range2 check_object)
{

int function;

    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
}

int  check_force (num_range2 check_object)
{

int function;

    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
}

int  check_smartness (num_range2 check_object)
{
```

```
int function;

    if (check_object<=100&&check_object>=0)
     function = TRUE;
    else
     function = FALSE;

    return(function);
}

error_num check_record (num_range1 *number_c, char *sex_c, struct date *birthdate_c,
num_range2 *experience_c, num_range2 *force_c, num_range2 *smartness_c)
{

int err_p;
error_num function;

    err_p = FALSE;
    if ((err_p == FALSE))
     if (check_number(*number_c))
         err_p = FALSE;
     else  {
         err_p = TRUE;
         function = 1;
     }
    if ((err_p == FALSE))
     if (check_sex(*sex_c))
         err_p = FALSE;
     else  {
         err_p = TRUE;
         function = 2;
     }
    if ((err_p == FALSE))
     if (check_birthdate(*birthdate_c))
         err_p = FALSE;
     else  {
         err_p = TRUE;
         function = 3;
```

```
}
if ((err_p == FALSE))
    if (check_experience(*experience_c))
        err_p = FALSE;
    else  {
        err_p = TRUE;
     function = 1;
    }
if ((err_p =- FALSE))
 if (check_force(*force_c))
     err_p = FALSE;
    else  {
     err_p = TRUE;
         function - 5;
        }
    if ((err_p == FALSE))
        if (check_smartness(*smartness_c))
            err_p = FALSE;
     else  {
            err_p = TRUE;
        function = 6;
     }
    if (err_p == FALSE)
     function = 0;

    return(function);
}

int  condition_select (char *condition_message)
{

int condition_code;
int function;

    printf("\n");
    printf("%s\n",condition_message);
    printf("          1 record 2 No 3 name 4 sex\n");
```

```
    printf("            5 birthday 6 experience 7 force 8 smartness\n");

    printf("\n");
    printf("choise:(1~8)");
    scanf("%d",&condition_code);
    function = condition_code;

    return(function);
}

void output_title(void)
{
    printf("\n");
    printf("  record  no  name  sex  birthday   experience  force   smartness\n");


}

void output_it (struct node *c,int n)
{

    printf("%7d",n);     //数据输出
    printf("%7d",c->number);
    printf("%10s",c->name);
    printf("%6c",c->sex);
    printf("%6d/%2d/%2d",c->birthdate.da_year,c->birthdate.da_mon,
          c->birthdate.da_day);
    printf("%10d",c->experience);
    printf("%10d",c->force);
    printf("%10d",c->smartness);
    printf("\n");

}

void modify_it (struct node *c,struct node *l,int n)
{
```

```
printf("modify %d input -1 ,still\n",n);
 printf("record %d\n",n);
 printf("No:");
 scanf("%d",&number_c);
 printf("name:");
 gets(name_c);
 gets(name_c);
 printf("sex(M,F):");
 scanf("%c",&sex_c);
 printf("birthda_day(1978 16 1):");
 scanf("%d %d %d",&birthdate_c.da_year,&birthdate_c.da_day,&birthdate_c.da_mon);
 printf("experience:");
 scanf("%d",&experience_c);
 printf("force:");
 scanf("%d",&force_c);
 printf("smartness:");
 scanf("%d",&smartness_c);
 printf("\n");
if (number_c != - 1)
 if (check_number(number_c))
     c->number = number_c;
 else
     printf("out record\n");
if (strcmp(name_c,"-1") != 0)
 strcpy (c->name,name_c);
if (sex_c != '-')
 if (check_sex(sex_c))
     c->sex = sex_c;
 else
     printf("sex error\n");
if ((birthdate_c.da_year != - 1) && (birthdate_c.da_mon != - 1) &&
     (birthdate_c.da_day != - 1))
 if (check_birthdate(birthdate_c))
     c->birthdate = birthdate_c;
 else
     printf("birthday error\n");
if (experience_c != - 1)
 if (check_experience(experience_c))
```

```
        c->experience - experience_c;
    else
        printf("experience error\n");
    if (force_c !=  - 1)
    if (check_force(force_c))
        c->force = force_c;
    else
        printf("force error\n");
    if (smartness_c !=  - 1)
    if (check_smartness(smartness_c))
        c->smartness = smartness_c;
    else
        printf("smartness error\n");
    }

    void swap_it (struct node **current,struct node **last,struct node **c_bak,struct node
**l_bak)
    {
    //用于排序

    struct node *tmp_c,
            *tmp_l;
    //使C_BAK总是指向链表尾部
        if (((*current)->front->front == nil) && (*last == nil)) {
           *current = (*current)->front;
            head = *c_bak;
            (*current)->front = head;
            (*c_bak)->next = *current;
     (*current)->next = nil;
        *last = nil;
     *c_bak = *current;
     *l_bak = nil;
    }
    else if (((*current)->front->front != nil) && (*last == nil)) {
        *current = (*current)->front;
        (*current)->front->next = *c_bak;
        (*c_bak)->front = (*current)->front;
```

```
    (*c_bak)->next = *current;
    (*current)->front = *c_bak;
    (*current)->next = nil;
    *last = (*current)->next;
*c_bak = *current;
    *l bak = *last;
}
else if (((*current)->front->front == nil) && (*last != nil)) {
    head = *current;
    *current = (*current)->front;
    head->next->front = *current;
(*current)->next = head->next;
    head->front = nil;
head->next = *current,
    (*current)->front = head;
*last = (*current)->next;
}
else {
tmp_c = *current;
    *current = (*current)->front;
    (*current)->front->next = tmp_c;
    tmp_c->next->front = *current;
    tmp_c->front = (*current)->front;
    (*current)->next = tmp_c->next;
    tmp_c->next = *current;
    (*current)->front = tmp_c;
}

}

}

void delete it (struct node **current, struct node  **last)
{
//可以自动后移指针

struct node *tmp;

    if ((*current == head) && (*last == nil)) {
        free(*current);
```

```
    head = nil;
*current = nil;
    *last = nil;
    count = 0;
}
else if ((*current != head) && (*last == nil)) {
    *current = (*current)->front;
free(*current);
    (*current)->next = *last;
    count = count - 1;
}
else if ((*current == head) && (*last != nil)) {
    head = *last;
free(*current);
    *current = head;
    (*current)->front = nil;
*last = (*current)->next;
    count = count - 1;
}
else  {
    (*current)->front->next = (*current)->next;
    (*current)->next->front = (*current)->front;
    tmp = *current;
    *current = *last;
    *last = (*current)->next;
free(tmp);
    tmp = nil;
        count = count - 1;
    }

}

void add_record (void)
{

struct node *tmp;

    if (head == nil) {
```

```
tmp = (stu_p) malloc (sizeof(struct node));
head = tmp;
head >next = nil;
current = head;
current->front = nil;
last = current->next;
}
else  {
tmp = (stu_p) malloc (sizeof(struct node));
    tmp->next = nil;
current->next = tmp;
    tmp->front = current;
 current = tmp;
    last = current->next;
}    //数据赋值
current->number = number_c;
strcpy (current->name,name_c);
current->sex = sex c;
current->birthdate = birthdate c;
current->experience = experience_c;
current->force = force_c;
current->smartness = smartness_c;
count = count + 1;

}

void insert_it (struct node **c,int n,int *p)
{

struct node *insert;
error_num err;

    *p = 0;
 printf("record %d\n",n);
 printf("No:");
 scanf("%d",&number_c);
 printf("name:");
 gets(name_c);
```

```
gets(name_c);
printf("sex(M,F):");
scanf("%c",&sex_c);
printf("birthda day(1978 16 1):");
scanf("%d %d %d",&birthdate_c.da_year,&birthdate_c.da_day,&birthdate_c.da_mon);
printf("experience:");
scanf("%d",&experience_c);
printf("force:");
scanf("%d",&force_c);
printf("smartness:");
scanf("%d",&smartness c);
printf("\n");
err = check_record(&number_c,&sex_c,&birthdate_c,&experience_c,
    &force_c,&smartness_c);
switch (err) {
    case 0:
     add_record();
     n = n + 1;
     *p=1;
     break;
    case 1:
     printf("no wrong:1-1000\n");

     printf("try again\n");
     break;
    case 2:
     printf("sex wrong:M male F female\n");

     printf("try again\n");
     break;
    case 3:
     printf("birthday wrong:1978 1 16\n");

     printf("try again\n");
     break;
    case 4:
     printf("experience wrong:0-100\n");
```

```
        printf("try again\n");
        break;
      case 5:
        printf("force wrong:0-100\n");

        printf("try again\n");
        break;
      case 6:
        printf("smartness wrong:0-100\n");

        printf("try again\n");
        break;
    }
  if (*p == 1) {
      if (*c == nil) {
          add_record();
  }
  else if (*c == head) {
          insert = (stu_p) malloc (sizeof(struct node));
          insert->front = nil;
          insert->next = nil;
      head->front = insert;
          insert->next = head;
      head = insert;
          insert->number = number_c;
          strcpy (insert->name,name_c);
          insert->sex = sex_c;
          insert->birthdate = birthdate_c;
          insert->experience = experience_c;
   insert->force = force_c;
          insert->smartness = smartness_c;
          count = count + 1;
  }
  else  {
      insert = (stu_p) malloc (sizeof(struct node));
          insert->front = nil;
   insert->next = nil;
          (*c)->front->next = insert;
```

```
            insert >front = (*c)->front;
            (*c)->front = insert;
            insert->next = *c;
        insert->number = number_c;
            strcpy (insert->name, name_c);
        insert->sex = sex_c;
            insert->birthdate = birthdate_c;
            insert->experience = experience_c;
            insert->force = force_c;
            insert->smartness = smartness_c;
            count = count + 1;
    }
}

}

void save_record (void)
{

int n;
char path[81];
struct node *c,
            *l;

    if (head == nil)
     printf("no record\n");
    else  {
     n = 0;     //文件初始化
     printf("\n");
     printf("write file name ps:pb.TXT:\n");
     gets(path);
     db_file=fopen(path,"wt");     //指针初始化
     c = head;
     l = head->next;
     fprintf(db_file,"%d\n",count);     //写入最大记录数
     while (c != nil) {     //数据输出
         fprintf(db_file,"%d",c->number);
         fprintf(db_file,"%10s",c->name);
```

```
        fprintf(db_file,"%2lc",c->sex);
        fprintf(db_file,"%5d%3d%3d",c->birthdate.da_year,
             c->birthdate.da_day,c->birthdate.da_mon);
        fprintf(db_file,"%4d",c->experience);
        fprintf(db_file,"%4d",c->force);
        fprintf(db_file,"%4d",c->smartness);
        fprintf(db_file,"\n");
           n = n + 1;
           c = l;
           l = c->next;
       }
    fclose(db_file);    //保护代码
    if (n != count)
        printf("sum of record wrong\n");
     printf("have been saved\n");
    }

}

void close_database (int n)
{
//N为内部使用,信息显示开关

struct node *c,
           *l;
int i;
char pan;

    if (head != nil) {
    c = head;
    l = c->next;
    pan = ' ';
    if (n == 1) {
        printf("\n");
        printf("save database?(Y/N)\n");
        scanf("%c",&pan);
        if (pan == 'Y')
              save_record();
```

```
            printf("\n");
        }
        for (i = 1; i <= count; i++)
            delete_it(&c,&l);
        head = nil;
        current = nil;
        last = nil;
        count = 0;
        if (n == 1)
         printf("database close\n");
    }

    void output_record (void)
    {

    int n,
        i;
    struct node *c,
                *l;

        if (head == nil)
         printf("no record\n");
        else  {
            n = 1;    //指针初始化
            c = head;
            l = head->next;
            output_title();
            for (i = 1; i <= count; i++) {
                output_it(c,n);
                n = n + 1;
                c = l;
             l = c->next;
            }
        }

    }
```

```
void del_record(void)
{

int condition_code,
int i,
    deleted_num;

int long_c,long_d,
int del_num;
num_rangel del_number;
char del_name[11];
char del_sex;
struct date del_birthdate;
num range2 del experience,
num_range2 del_force;
num_range2 del_smartness;
struct node *c_bak,
            *l_bak;

    condition_code = condition_select("del condition:");
    c_bak = current;
    l_bak = last;
    switch (condition_code) {
     case 1:
         printf("\n");
         printf("record:");
         scanf("%d",&del_num);
         if ((del_num >= 1) && (del_num <= count)) {
          if (head != nil) {
                current = nil;
        last = nil;
                for (i = 1; i <= del_num; i++) {
         if (i == 1) {
             current = head;
             last = current->next;
         }
         else  {
             current = last;
```

```
                last = current->next;
            }
            }
            delete it(&current,&last);
            printf(" record %d delete\n",del_num);
        }
        else
            printf("no record like u need\n");
        }
        else
        printf("out record\n");
        break;
case 2:
        printf("\n");
        printf("no:");
        scanf("%d",&del_number);
        if ((del_number >= 1) && (del_number <= 1000)) {
        if (head != nil) {
                current = head;
                last = current->next;
            deleted_num = 0;
                for (i = 1; i <= count; i++) {
            if (current->number == del_number) {
                        delete_it(&current,&last);
                deleted_num = deleted_num + 1;
            }
                    else {
                current = last;
                last = current->next;
            }
            }
            if (deleted_num != 0)
            printf(" record %d delete\n",deleted_num);
            else
            printf("no record like u need\n");
        }
        else
            printf("no record like u need\n");
```

```
        }
        else
         printf("out record\n");
        break;
    case 3:
        printf("\n");
        printf("name:");
        gets(del_name);
        if (head != nil) {
         current = head;
         last = current->next;
         deleted_num = 0;
         for (i = 1; i <= count; i++) {
             if (strcmp(current->name, del_name) == 0) {
              delete_it(&current, &last);
              deleted_num = deleted_num + 1;
             }
             else  {
              current = last;
              last = current->next;
             }
         }
         if (deleted_num != 0)
             printf(" record %d delete\n", deleted_num);
         else
             printf("no record like u need\n");
        }
        else
         printf("out record\n");
        break;
    case 4:
           printf("\n");
        printf("sex:");
        scanf("%c", &del_sex);
        if (head != nil) {
         current = head;
         last = current->next;
         deleted_num = 0;
```

```
            for (i - 1; i <= count; i++) {
                if ((current->sex == del_sex)) {
                    delete_it(&current,&last);
                    deleted_num = deleted_num + 1;
                }
            else  {
             current - last;
             last = current->next;
            }
        }
        if (deleted_num != 0)
            printf(" record %d delete\n",deleted_num);
        else
             printf("no record like u need\n");
            }
            else
             printf("out record\n");
            break;
        case 5:
            printf("\n");
            printf("birthday:");
            scanf("%d%d%d",&del_birthdate.da_year,&del_birthdate.da_mon,
                &del_birthdate.da_day);
            if (head != nil) {
             current = head;
             last = current->next;
             deleted_num = 0;
             long_d = del_birthdate.da_year;
             long_d = long_d * 100;
             long_d = long_d + del_birthdate.da_mon;
             long_d = long_d * 100;
             long_d = long_d + del_birthdate.da_day;
             for (i = 1; i <= count; i++) {
                long_c = current->birthdate.da_year;
                long_c = long_c * 100;
                long_c = long_c + current->birthdate.da_mon;
                long_c = long_c * 100;
                long_c = long_c + current->birthdate.da_day;
```

```
    if ((long_c == long_d)) {
     delete_it(&current,&last);
     deleted_num = deleted_num - 1;
    }
    else  {
     current = last;
     last = current->next;
    }
}
if (deleted_num != 0)
    printf(" record %d delete\n",deleted_num);
else
    printf("no record like u need\n");
}
else
 printf("out record\n");
 break;
case 6:
    printf("\n");
 printf("experience:");
 scanf("%d",&del_experience);
 if (head != nil) {
  current = head;
  last = current->next;
  deleted_num = 0;
  for (i = 1; i <= count; i++) {
      if ((current->experience == del_experience)) {
       delete_it(&current,&last);
       deleted_num = deleted_num + 1;
            }
      else  {
       current = last;
       last = current->next;
      }
}
if (deleted_num != 0)
    printf(" record %d delete\n",deleted_num);
else
```

```
            printf("no record like u need\n");
        }
        else
         printf("out record\n");
        break;
    case 7:
        printf("\n");
        printf("force:");
        scanf("%d",&del_force);
           if (head != nil) {
         current = head;
                last = current->next;
         deleted_num = 0;
          deleted_num = 0;
              for (i = 1; i <= count; i++) {
                  if ((current->force == del_force)) {
                      delete_it(&current,&last);
           deleted_num = deleted_num + 1;
          }
          else  {
                      current = last;
           last = current->next;
          }
     }
     if (deleted_num != 0)
        printf(" record %d delete\n",deleted_num);
     else
        printf("no record like u need\n");
    }
    else
        else
         printf("out record\n");
        break;
    case 8:
        printf("\n");
        printf("smartness:");
        scanf("%d",&del_smartness);
           if (head != nil) {
         current = head;
                last = current->next;
```

```
deleted_num - 0;
        for (i = 1, i <= count; i++) {
            if ((current->smartness == del_smartness)) {
     delete_it(&current,&last);
                deleted_num = deleted_num + 1;
    }
    else  {
                current = last;
     last = current->next;
    }
};
if (deleted_num != 0)
    printf(" record %d delete\n",deleted_num);
     else
         printf("no record like u need\n");
        }
        else
         printf("out record\n");
        break;
 default:
        printf("delete error\n");
        break;
}
if (last != nil) {
 current = c_bak;
 last = l_bak;
}

}

}

void find_record (void)
{

int condition_code;
int i,
    found_num;

int long_c, long_f;
```

```
struct node *c,
            *l;
int find_num;
num_range1 find number,
char find_name[11];
char find_sex;
struct date find birthdate;
num_range2 find_experience;
num_range2 find_force;
num_range2 find_smartness;

    condition_code = condition_select("find condition:");
switch (condition code) {
casc 1:
    printf("\n");
    printf("record:");
    scanf("%d",&find_num);
    if ((find_num >= 1) && (find_num <= count)) {
          if (head != nil) {
              c = nil;
              l = nil;
              found_num = 1;
        for (i = 1; i <= find_num; i++) {
                  if (i == 1) {
                      c = head;
                      l = c->next;
                  }
                  else {
                      c = l;
                      l = c->next;
                  }
              }
          output_title();
    output_it(c, find_num);
    printf("find %d\n", found_num);
}
else
    printf("no record\n");
```

```
}
    else
 printf("out record\n");
 break;
case 2:
    printf("\n");
 printf("no:");
 scanf("%d",&find_number);
 if (head != nil) {
        c = head;
        l = c->next;
        found_num = 0;
          output_title();
          for (i = 1; i <= count; i++) {
              if (c->number == find_number) {
       output_it(c,i);
                found_num = found_num + 1;
       c = l;
                c = c->next;
      }
      else  {
                c = l;
                l = c->next;
      }
 }
 if (found_num != 0)
      printf("find %d\n",found_num);
       else
          printf("no record\n");
      }
      else
       printf("out record\n");
      break;
case 3:
      printf("\n");
      printf("name:");
      gets(find_name);
      if (head != nil) {
```

```
c = head;
l = c->next;
      found_num = 0;
      output_title();
      for (i = 1; i <= count; i++) {
          if (strcmp(c->name, find_name) == 0) {
     output_it(c, i);
              found_num = found_num + 1;
     c = l;
     c = c->next;
    }
    else {
         c = l;
         l = c->next;
        }
     }
     if (found_num != 0)
         printf("find %d\n", found_num);
     else
         printf("no record\n");
    }
    else
     printf("out record\n");
    break;
case 4:
    printf("\n");
    printf("sex:");
    scanf("%c", &find_sex);
    if (head != nil) {
        c = head;
l = c->next;
      found_num = 0;
      output_title();
      for (i = 1; i <= count; i++) {
          if (c->sex == find_sex) {
     output_it(c, i);
              found_num = found_num + 1;
     c = l;
```

```
                c = c->next;
        }
        else  {
         c = l;
         l = c->next;
        }
    }
    if (found_num != 0)
        printf("find %d\n", found_num);
    else
        printf("no record\n");
}
        else
         printf("out record\n");
        break;
case 5:
        printf("\n");
        printf("birthday:");
        scanf("%d%d%d", &find_birthdate.da_year, &find_birthdate.da_mon,
            &find_birthdate.da_day);
        if (head != nil) {
         c = head;
         l = c->next;
         found_num = 0;
                output_title();
         long_f = find_birthdate.da_year;
                long_f = long_f * 100;
         long_f = long_f + find_birthdate.da_mon;
         long_f = long_f * 100;
         long_f = long_f + find_birthdate.da_day;
         for (i = 1; i <= count; i++) {
            long_c = c->birthdate.da_year;
                long_c = long_c * 100;
            long_c = long_c + c->birthdate.da_mon;
                long_c = long_c * 100;
            long_c = long_c + c->birthdate.da_day;
                if (long_c == long_f) {
             output_it(c, i);
```

```
                found_num = found_num + 1;
                c = 1;
        c = c >next;
       }
            else
        c = 1;
                l = c >next;
            }
         }
if (found_num != 0)
    printf("find %d\n", found_num);
else
     printf("no record\n");
}
else
 printf("out record\n");
break;
case 6:
    printf("\n");
printf("experience:");
scanf("%d",&find_experience),
    if (head != nil) {
        c = head;
        l = c->next;
        found_num = 0,
 output_title();
 for (i = 1; i <= count, i++) {
            if (c->experience == find_experience) {
                output_it(c, i);
                found_num = found_num + 1;
        c = l;
        l = c->next;
       }
       else  {
        c = l;
        l = c->next;
       }
}
```

```
        if (found num != 0)
            printf("find %d\n", found_num);
        else
            printf("no record\n");
       }
       else
        printf("out record\n");
       break;
   case 7:
       printf("\n");
       printf("force:");
       scanf("%d", &find_force);
       if (head != nil) {
        c = head;
        l = c->next;
        found_num = 0;
        output_title();
        for (i = 1; i <= count; i++) {
            if (c->force == find_force) {
             output_it(c, i);
             found_num = found_num + 1;
             c = l;
             l = c->next,
            }
            else  {
             c = l;
             l = c->next;
            }
        }
        if (found_num != 0)
            printf("find %d\n", found_num);
        else
            printf("no record\n");
       }
       else
        printf("out record\n");
       break;
   case 8:
```

```
printf("\n");
printf("smartness:");
scanf("%d",&find_smartness);
if (head != nil) {
 c = head;
 l = c->next;
        found_num = 0;
        output_title();
        for (i = 1; i <= count; i++) {
      if (c->smartness == find_smartness) {
       output_it(c,i);
                found_num = found_num + 1;
       c = l;
       l = c->next;
      }
      else {
       c = l;
       l = c->next;
      }
 }
 if (found_num != 0)
     printf("find %d\n",found_num);
 else
     printf("no record\n");
}
   else
 printf("out record\n");
break;
      default:
          printf("find error\n");
          break;
      }

}

void sort_record (void)
{
```

```
int condition_code;
int i,
    j;

int long_c, long_f;
struct node *c_bak,
        *l_bak;

    condition_code = condition_select("sort condition:");
    switch (condition_code) {
     case 1:
         printf("record over\n");
         break;
case 2:
    if ((head != nil) && (head->next != nil)) {
     c_bak = current;
     l_bak = last;
     for (i = count - 1; i >= 1; i--) {
         current = c_bak;
         last = l_bak;
         for (j = 1; j <= i; j++) {
          if (current->number < current->front->number)
              swap_it(&current, &last, &c_bak, &l_bak);
                   current = current->front;
          last = current->next;
                }
           }
           current = c_bak;
           last = l_bak;
     }
     printf("no over\n");
     break;
case 3:
    if ((head != nil) && (head->next != nil)) {
         c_bak = current;
         l_bak = last;
         for (i = count - 1; i >= 1; i--) {
             current = c_bak;
```

```
                last = l_bak;
                for (j = 1; j <= i; j++) {
                    if (strcmp(current->name,
              current->front->name) < 0)
                        swap_it(&current,&last,&c_bak,&l_bak);
                    current = current->front;
          last = current->next;
                }
            }
            current = c_bak;
            last = l_bak;
        }
    printf("name over\n");
        break;
    case 4:
     if ((head != nil) && (head->next != nil)) {
            c_bak = current;
            l_bak = last;
            for (i = count - 1; i >= 1; i--) {
                current = c_bak;
                last = l_bak;
                for (j = 1; j <= i; j++) {
                    if (current->sex < current->front->sex)
                        swap_it(&current,&last,&c_bak,&l_bak);
                    current = current->front;
                    last = current->next;
        }
     }
            current = c_bak;
            last = l_bak;
        }
     printf("sex over\n");
     break;
    case 5:
        if ((head != nil) && (head->next != nil)) {
            c_bak = current;
            l_bak = last;
            for (i = count - 1; i >= 1; i--) {
```

```
            current = c_bak;
        last = l_bak;
            for (j = 1; j <= i; j++) {
        long_c = current->birthdate.da_year;
                long c = long_c * 100;
        long_c = long_c + current->birthdate.da_mon;
                long c = long_c * 100;
        long_c = long_c + current->birthdate.da_day;
    long_f = current->front->birthdate.da_year;
                long_f = long_f * 100;
                long_f = long_f +
            current->front->birthdate.da_mon;
                long_f = long_f * 100;
            long_f = long_f + current->front->birthdate.da_day;

            if (long_c < long_f)
                swap_it(&current,&last,&c_bak,&l_bak);
            current = current->front;
            last = current->next;
           }
     }
     current = c_bak;
     last = l_bak;
    }
    printf("birthday over\n");
    break;
   case 6:
       if ((head != nil) && (head->next != nil)) {
    c_bak = current;
           c_bak = current;
           l_bak = last;
           for (i = count - 1; i >= 1; i--) {
               current = c_bak;
               last = l_bak;
               for (j = 1; j <= i; j++) {
                   if (current->experience < current->front->experience)
                       swap_it(&current,&last,&c_bak,&l_bak);
                   current = current->front;
         last = current->next;
```

```
                }
        }
                current = c_bak;
                last = l_bak;
            }
        printf("experience over\n");
        break;
    case 7:
            if ((head != nil) && (head->next != nil)) {
                c_bak = current;
        l_bak = last;
        for (i = count - 1; i >= 1; i--) {
                    current = c_bak;
                    last = l_bak;
                    for (j = 1; j <= i; j++) {
                        if (current->force < current->front->force)
                            swap_it(&current,&last,&c_bak,&l_bak);
                        current = current->front;
                        last = current->next;
                    }
                }
         current = c_bak;
                last = l_bak;
        }
        printf("force over\n");
        break;
    case 8:
            if ((head != nil) && (head->next != nil)) {
                c_bak = current;
                l_bak = last;
    for (i = count - 1; i >= 1; i—) {
                    current = c_bak;
                    last = l_bak;
                    for (j = 1; j <= i; j++) {
                        if (current->smartness < current->front->smartness)
                  swap_it(&current,&last,&c_bak,&l_bak);
                        current = current->front;
                        last = current->next;
```

```
                    ;
                }
                current = c_bak;
            last = l_bak;
              }
            printf("smartness over\n");
            break;
        }

}

void modify_record (void)
{

int condition_code;
int i,
    modified_num;


int long_c, long_m;
struct node *c,
        *l;
int mdf_num;
num_range1 mdf_number;
char mdf_name[11];
char mdf_sex;
struct date mdf_birthdate;
num_range2 mdf_experience;
num_range2 mdf_force;
num_range2 mdf_smartness;

    condition_code = condition_select("modify:");
    switch (condition_code) {
     case 1:
         printf("\n");
         printf("record:");
         scanf("%d", &mdf_num);
            if ((mdf_num >= 1) && (mdf_num <= count)) {
```

```
if (head != nil) {
        c = nil;
        l = nil;
        for (i = 1; i <= mdf_num; i++) {
    if (i == 1) {
            c = head;
            l = c->next;
    }
        else  {
            c = l;
        l = c->next.
    }
            }
            modify_it(c, l, i);
        printf("%d modify\n", mdf_num);
     }
     else
        printf("no record\n");
    }
       else
        printf("out record\n");
    break;
case 2:
    printf("\n");
    printf("no:");
    scanf("%d", &mdf_number);
       if (head != nil) {
c = head;
l = c->next;
modified_num = 0;
for (i = 1; i <= count; i++) {
    if (c->number == mdf_number) {
     modify_it(c, l, i);
     modified_num = modified_num + 1;
     c = l;
     c = c->next;
    }
    else  {
```

```
            c = l;
            l = c->next;
           }
        }
        if (modified_num != 0)
            printf(" %d modify\n",modified_num);
        else
            printf("no record\n");
       }
       else
            printf("no record\n");
       break;
case 3:
       printf("\n");
       printf("name:");
       gets(mdf_name);
       if (head != nil) {
        c = head;
        l = c->next;
        modified_num = 0;
        for (i = 1; i <= count; i++) {
                  if (strcmp(c->name,mdf_name) == 0) {
                      modify_it(c,l,i);
            modified_num = modified_num + 1;
                      c = l;
            c = c->next;
           }
           else {
            c = l;
            l = c->next;
           }
        }
        if (modified_num != 0)
            printf(" %d modify\n",modified_num);
        else
            printf("no record\n");
       }
       else
```

```
            printf("no record\n");
        break;
    case 4:
        printf("\n");
        printf("sex:");
        scanf("%c",&mdf_sex);
        if (head != nil) {
         c = head,
         l = c->next;
         modified_num = 0;
         for (i = 1; i <= count; i++) {
             if (c->sex == mdf_sex) {
         modify_it(c,l,i);
         modified_num = modified_num + 1;
                  c = l;
         c = c->next;
               }
               else  {
                   c = l;
                   l = c->next;
        }
 }
 if (modified_num != 0)
     printf(" %d modify\n",modified_num);
 else
     printf("no record\n");
}
else
            printf("no record\n");
        break;
    case 5:
        printf("\n");
        printf("birthday:");
        scanf("%d%d%d",&mdf_birthdate.da_year,&mdf_birthdate.da_mon,
            &mdf_birthdate.da_day);
        if (head != nil) {
         c = head;
               l = c->next;
```

```
        modified_num = 0;
long_m = mdf_birthdate.da_year;
        long_m = long_m * 100;
long_m = long_m + mdf_birthdate.da_mon;
        long_m = long_m * 100;
long_m = long_m + mdf_birthdate.da_day;
for (i = 1; i <= count; i++) {
    long_c = c->birthdate.da_year;
            long_c = long_c * 100;
    long_c = long_c + c->birthdate.da_mon;
    long_c = long_c * 100;
    long_c = long_c + c->birthdate.da_day;
    if ((long_c == long_m)) {
                modify_it(c,1,i);
     modified_num = modified_num + 1;
                c = 1;
                c = c->next;
            }
            else  {
                c = 1;
                1 = c->next;
            }
}
       if (modified_num != 0)
    printf(" %d modify\n",modified_num);
 else
    printf("no record\n");
}
else
    else
        printf("no record\n");
    break;
case 6:
    printf("\n");
    printf("experience:");
    scanf("%d",&mdf_experience);
       if (head != nil) {
            c = head;
            1 = c->next;
```

```
        modified_num = 0;
        for (i = 1; i <= count; i++) {
            if (c->experience == mdf_experience) {
    modify_it(c, l, i);
    modified_num = modified_num + 1;
    c = l;
    c = c->next;
    }
    else {
    c = l;
    l = c->next;
    }
        }
        if (modified_num != 0)
            printf(" %d modify\n", modified_num);
        else
            printf("no record\n");
        }
        else
            printf("no record\n");
        break;
    case 7:
        printf("\n");
        printf("force:");
        scanf("%d", &mdf_force);
        if (head != nil) {
        c = head;
        l = c->next;
        modified_num = 0;
        for (i = 1; i <= count; i++) {
            if (c->force == mdf_force) {
            modify_it(c, l, i);
            modified_num = modified_num + 1;
            c = l;
            c = c->next;
            }
            else {
            c = l;
```

```
            l = c->next;
           }
        }
        if (modified_num != 0)
            printf(" %d modify\n",modified_num);
        else
            printf("no record\n");
       }
       else
            printf("no record\n");
       break;
case 8:
printf("\n");
printf("smartness:");
scanf("%d",&mdf_smartness);
   if (head != nil) {
c = head;
       l = c->next;
modified_num = 0;
       for (i = 1; i <= count; i++) {
           if (c->smartness == mdf_smartness) {
               modify_it(c,l,i);
               modified_num = modified_num + 1;
               c = l;
    c = c->next;
           }
           else {
               c = l;
               l = c->next;
    }
 }
 if (modified_num != 0)
     printf(" %d modify\n",modified_num);
 else
     printf("no record\n");
}
else
     printf("no record\n");
```

```
            break;
        default:
            printf("modify error\n");
            break;
    }

}

void read_record (void)
{

int n,i,len;
char *path;
char panduan;

    count = 0;     //文件初始化
    printf("\n");
    printf("give the name of dbf like pb.txt:\n");
    gets(path);
    if((db_file=fopen(path,"rt"))==NULL)
     printf("this file is not exist");
    else
    {
    fseek(db_file,0,SEEK_SET);
    fscanf(db_file,"%d",&n);     //读取最大记录数
    for (i = 1; i <=n; i++) {     //数据载入缓冲
     fscanf(db_file,"%d",&number_c);
     fscanf(db_file,"%10s",&name_c);
     fseek(db_file,1,SEEK_CUR);
     fscanf(db_file,"%c",&sex_c);
     fscanf(db_file,"%5d",&birthdate_c.da_year);
     fscanf(db_file,"%3d",&birthdate_c.da_day);
     fscanf(db_file,"%3d",&birthdate_c.da_mon);
     fscanf(db_file,"%4d",&experience_c);
     fscanf(db_file,"%4d",&force_c);
     fscanf(db_file,"%4d",&smartness_c);
     add_record();
    }
```

```
    printf("file read end\n");
    printf("record number is %d \n",i-1);
    }

}

void input_record (void)
{

int n,err;

    n = count + 1;
    printf("\n");
    printf("after enter\n");
    number_c=0;
    while ( number_c!=-1 ) {
     printf("record %d\n",n);
     printf("No:");
     scanf("%d",&number_c);
     printf("name:");
     gets(name_c);
     gets(name_c);
     printf("sex(M,F):");
     scanf("%c",&sex_c);
     printf("birthda_day(1978 16 1):");
     scanf("%d %d %d",&birthdate_c.da_year,&birthdate_c.da_day,&birthdate_c.da_mon);
     printf("experience:");
     scanf("%d",&experience_c);
     printf("force:");
     scanf("%d",&force_c);
     printf("smartness:");
     scanf("%d",&smartness_c);
     printf("\n");
     err = check_record(&number_c,&sex_c,&birthdate_c,&experience_c,
         &force_c,&smartness_c);
     switch (err) {
        case 0:
         add_record();
```

```
            n = n + 1;
            break;
          case 1:
            printf("no wrong:1-1000\n");

            printf("try again\n");
            break;
          case 2:
            printf("sex wrong:M male F female\n");

            printf("try again\n");
            break;
          case 3:
            printf("birthday wrong:1978 1 16\n");

            printf("try again\n");
            break;
          case 4:
            printf("experience wrong:0-100\n");

            printf("try again\n");
            break;
          case 5:
            printf("force wrong:0-100\n");

            printf("try again\n");
            break;
          case 6:
            printf("smartness wrong:0-100\n");

            printf("try again\n");
            break;
        }
        printf("enter to continue,-1 to end");
        printf("\n");
      }

}
```

```
void insert_record (void)
{

int i,
    n,
    p;
struct node *c,
         *l;

    if (head != nil) {
        printf("\n");
     printf("insert record:");
     scanf("%d",&n);
     if ((n >= 1) && (n <= count)) {
         c = nil;
            l = nil;
            for (i = 1; i <= n; i++) {
                if (i == 1) {
                    c = head;
                    l = c->next;
                }
                else {
                    c = l;
                    l = c->next;
                }
            }
         insert_it(&c, n, &p);
         if (p == 1)
          printf("insert over\n");
         else
          printf("insert wrong\n");
     }
     else
         printf("out record\n");
}
else {
     insert_it(&head, 1, &p);
         if (p == 1)
```

```
            printf("insert over\n");
            else
            printf("insert wrong\n");
        }

}

void command_select (char command_line[])
{

    if ((strcmp(command_line,"APPEND") == 0) || (strcmp(command_line,
            "ADD") == 0)) {
     strcpy (command_line,"");
     input_record();
    }
    else if ((strcmp(command_line,"DEL") == 0) || (strcmp(command_line,
        "DELETE") == 0)) {
     strcpy (command_line,"");
     del_record();
    }
    else if (strcmp(command_line,"SAVE") == 0) {
     strcpy (command_line,"");
     save_record();
    }
    else if (strcmp(command_line,"LOAD") == 0) {
     strcpy (command_line,"");
     read_record();
    }
    else if (strcmp(command_line,"INSERT") == 0) {
     strcpy (command_line,"");
     insert_record();
    }
    else if ((strcmp(command_line,"SHOW") == 0) || (strcmp(command_line,
        "DIR") == 0)) {
     strcpy (command_line,"");
     output_record();
    }
    else if (strcmp(command_line,"FIND") == 0) {
```

```
 strcpy (command_line,"");
 find_record();
}
else if (strcmp(command_line,"SORT") == 0) {
 strcpy (command_line,"");
 sort_record();
}
else if (strcmp(command_line,"MODIFY") == 0) {
 strcpy (command_line,"");
 modify_record();
}
else if (strcmp(command_line,"HELP") == 0) {
 strcpy (command_line,"");

 printf("command help:");
 printf("APPEND,");
 printf("DELETE,");
 printf("SAVE,");
 printf("LOAD,");
 printf("INSERT,");
 printf("SHOW,");
 printf("FIND,");
 printf("SORT,");
 printf("CLOSE,");
 printf("MODIFY,");
 printf("QUIT,");
 printf("HELP");
}
else if (strcmp(command_line,"CLOSE") == 0) {
 strcpy (command_line,"");
    close_database(1);
}
else if ((strcmp(command_line,"QUIT") == 0) || (strcmp(command_line,
    "EXIT") == 0))
 strcpy (command_line,"QUIT");
else  {
 strcpy (command_line,"");
 printf("wrong command\n");
```

```
    }
}

void optimize(char command[])
{
char ch;
int i;

    ch = command[0];
    i = 1;
    while (ch != 0) {
     if ((ch > 96) && (ch < 123))
         command[i - 1] = ch - 32;
     i = i + 1;
     ch = command[i - 1];
    }

}

void main()
{
    head = nil;     //变量初始化
    current = nil;
    last = nil;
    count = 0;
    tmp = 0;
    strcpy (command_line,"");
    while (strcmp(command_line,"QUIT") != 0) {
        printf("\n");
        printf("PB.COMMAND>");
        gets(command_line);
        optimize(command_line);
        command_select(command_line);//命令调用
}
}
```

其效果如图 B-1 所示。

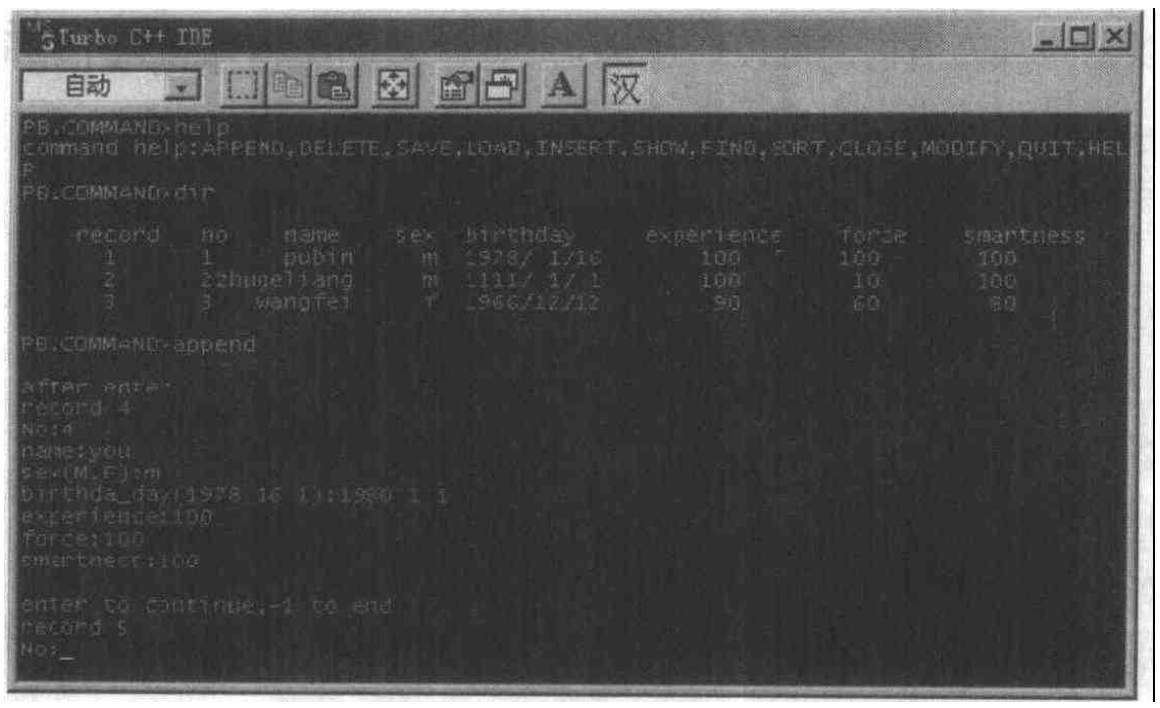


图 B-1

保存后数据文本文件中的内容：

```
4
1     pubin m 1978 16  1 100 100 100
2zhugeliang m 1111  1  1 100  10 100
3   wangfei f 1966 12 12  90  60  80
4       you m 1980  1  1 100 100 100
```

B.5　通用数据库设计

在这个简单数据库的设计基础上我们提出通用数据库设计的办法。通用数据库主要提供：

（1）自定义字段数目；

（2）自定义每个字段名、类型和长度；

为了实现以上功能的通用数据库，我们可通过建立多条链表组成一个链表矩表来实现。所谓链表矩阵就是让每一个字段托一条链表，而几个字段合并即成为一个链表集合。此种结构可以使我们方便的获得所有数据库数据、某一记录、某一字段数据和任意单元格数据。如图 B-2 所示。

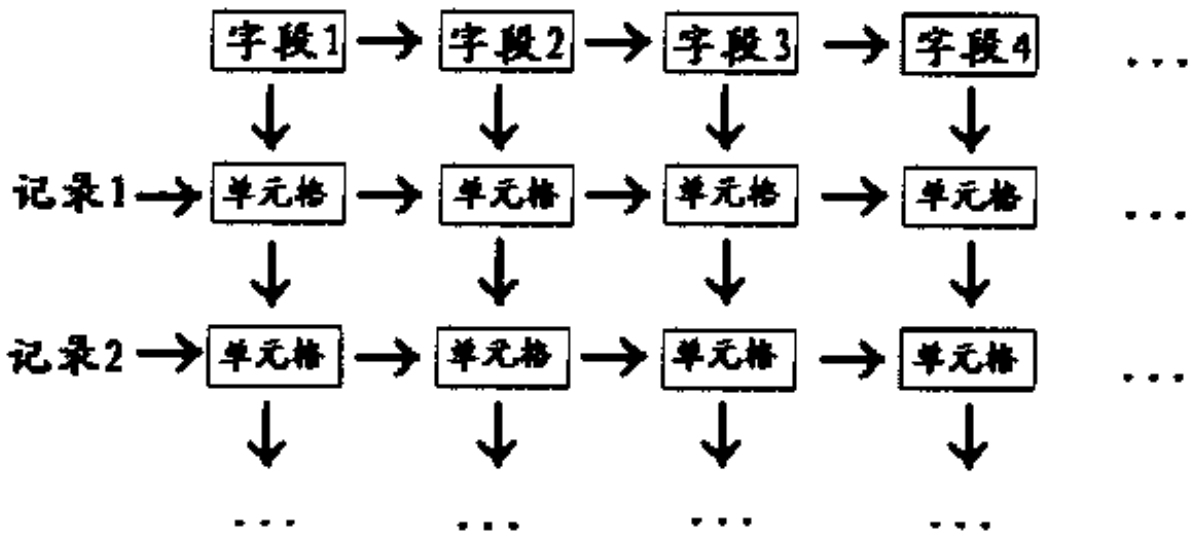


图 B-2

附录 B　简单数据库

为了实现以上的链表结构，可以通过建立一个字段名结构体和几个不同类型（数值、日期、字符串、逻辑和备注）的单元格（类型）结构体来完成，比如：

1. 字段名结构体的实现：

包含：

（1）字段名

（2）字段类型

（3）字段名宽度

（4）指向本字段类型的指针（5 个）

（5）字段名结构体指针

备注：字段名结构体乃数据链表之首，为了让他能适应各种数据类型环境，所以定义要非常全面。

2. 类型结构体的实现：

数值型

包含：

（1）数值

（2）数值型指针

字符型

包含：

（1）字符

（2）字符型指针

日期型

包含：

（1）年位

（2）月位

（3）日位

（4）日期型指针

逻辑型

包含：

（1）枚举逻辑位

（2）逻辑型指针

备注型

包含：

（1）字符串

（2）备注型指针

结构体定义如下：

字段名结构：

```
struct dbf {
    char name[20];//字段名称
```

```
    int type;//字段类型
    int width; //字段宽度
    struct dbf *next;//指向同一记录下一个字段单元格
   struct day_type *day_first;//指向日期结构
};
```

数值型结构

```
struct math_type{
long math_data;//数值类型
struct math_type *next;//指向下一个记录的同一字段单元格
}
```

字符型结构：

```
struct char_type{
char char_data;//字符类型
struct string_type *next;//指向下一个记录的同一字段单元格
}
```

日期型结构：

```
struct day_type{
struct date day_data;//日期类型
struct day_type *next;//指向下一个记录的同一字段单元格
}
```

逻辑型结构：

```
struct logic_type{
enum logic_data { true=1, false=0 };//逻辑类型
struct logic_type *next;//指向下一个记录的同一字段单元格
}
```

备注型结构：

```
struct string_type{
char *string_data;//字符串类型
struct string_type *next;//指向下一个记录的同一字段单元格
}
```

注释：

1.3.4.5.6.7. 此 6 个程序改自于 95-97 年软件报合订本；

2. 此函数和之后二维图形、图形文件、动画原理、子画面、接口技术章节中部分游戏函数改写于《电脑游戏编程技术大全》中 mfc 游戏函数，此处一并注释，之后不一一注释了；

8. 此程序改自于某中断方面书籍中的一个程序（书名忘记了并无法找到，抱歉）；

9. 此程序改自于张科技同学 pascal 程序。



北京希望电子出版社网站是一个完备的电子商务交易系统，集信息发布、客户服务、网上销售为一体，为广大读者和希望图书产品经销商提供全方位的服务。

信息发布——每日更新 在北京希望电子出版社的网站里有数千种图书光盘的数据资料，读者和经销商可以了解到图书光盘的所有信息，可以看到书的目录、内容提要、精彩章节，相关资料，也可以了解书的印刷、开本、定价等情况。如果您不知道想找的某一本书在什么地方，您可以在"书盘检索"的窗口里输入书名、作者、书号等信息查询。同时，网站数据每日更新，并发布本社经营活动的相关信息和业界动态。

技术支持——全程服务 技术支持是我社倡导"服务第一"理念的体现，在这里我们向读者介绍 IT 行业的新技术，新软件，新动态；组织大家在 BBS 讨论热点技术性问题，并有专家回答技术咨询。只要您是希望的用户，就可得到希望的技术支援，解除用户的后顾之忧。在这个栏目里您可看到大量精彩的计算机图形图像作品和 Flash 动画，此外，年轻的读者还可以在"读者俱乐部"发表自己的作品。

网上书店——方便快捷 在网站可以方便地选购图书光盘，付款方式灵活，如果您有招商银行的"一网通"，可以在网上付款。所有客户都可以享受到免费邮寄的服务。此外网上购书还有各种优惠，总有意想不到的惊喜等着您！

只要您点击 www.bhp.com.cn 决不会空手而归！

北京希望电子出版社网站

“书目信息”服务办法

为了帮助您尽早了解我社的新书出版信息，我社充分发挥自身计算机软件企业的技术优势，在本社的网站（www.bhp.com.cn）上开设了书目信息的服务项目，它可以使您每天与出版社同步得到新书出版信息，也可以为您提供本社数年来的全部书盘目录。

在网站查找、下载本社书盘目录的方法如下：

1、在网站的首页查找新书。在浏览器地址栏处输入 www.bhp.com.cn 之后便进入我社网站的首页，首页的右侧有“每周新书”栏目，该栏目每天更新，增添我社的当日新书。在首页上方的栏目条里，有“最近新书”栏目，包括近两个月的新书介绍。

2、在“书盘目录”栏目里，有本社图书光盘的分类目录。通过分类目录，可以找到同类图书关盘的目录和系列书目录，最后到达的是书的详细介绍，包括书的版权页信息、本书内容提要、目录等等。

3、在“经销商园地”栏目里，设有“书目下载”专栏。内容包括：1995 年——2000 年的书盘目录下载；1999 年书盘目录下载；2000 年书盘目录下载；2001 年内书盘目录按出版月份下载。书目下载是专门为希望电子图书经销商，书店提供的信息服务，与我社的图书目录单内容相同，是图书目录单的电子版，您可按此书目向我社定购图书。

书目下载文件为 zip 格式的压缩文档，解压缩后为 Word 格式。下载的书目操作步骤是：

（1）进入本社的网站（www.bhp.com.cn）“经销商园地”/“书目下载”栏目。

（2）点击所选取的连接，出现对话框，选取“将该文件保存到磁盘”，点击“确定”。

（3）在“另存为”对话框选择存放文件的路径，点击“保存”按钮后计算机便开始下载。

（4）下载完毕，启动自己的“zip 软件”，沿着前面文件的存放路径找到该文件，解开压缩，打开文件阅读。

如果您的计算机里还没有 zip 软件，可以在我们的网站下载一个。 地址是：首页/技术支持/工具箱/常用工具/ Winzip80

《希望书盘交流俱乐部》会员须知

一、 北京希望电子出版社的书盘立意新颖、风格迥异，受到广大读者喜爱。为了给长期热心支持和选购希望电脑书盘的朋友更多的回报，我社决定扩大《希望书盘交流俱乐部》，为此对读者入会条件和优惠政策作出如下调整：

1. 用户在本社一次性购买 100.00 元以上的书盘时，即可成为本俱乐部的会员，并在今后的购书中本市会员予以***八八折***优惠，外地会员购书**免加邮费同时优惠*九折*。**会员将在本俱乐部建有个人档案。
2. 会员购书时必须出示此卡以便打折并累计金额。外地会员邮购图书时请把卡号写在汇款单的附言条上，以便累计。
3. 俱乐部会员投稿优先刊登本社报刊。
4. 俱乐部不定期组织会员参加各项活动。
5. 会员可优先得到我社的新书资料和信息。
6. 我社长期征稿，欢迎所有会员投稿。（题目为“我（不）喜爱的一本希望图书”、“我读 XX 书后的感想”，或对我社的图书选题有何感想都可写稿寄来。如果来稿被采用，便予以刊登并同时得到一份纪念品。）

二、会员卡另有储值功能，会员可随意存储金额，如需订购图书只需拨打订购电话（010-62650298）经确认预留金额后即可发书。这项功能缩短了会员邮寄图书的周期，使会员在最短的时间内收到图书。会员可随时查询余额或续款。

三、会员若连续一年未购书盘，会员卡自动取消，需按入会资格重新入会。

四、会员卡遗失后，由该卡的指定联系人办理补卡手续。

五、持卡人或联系人的通讯地址及联系方法发生变动时，请及时与俱乐部联系。

声明：2000 年 12 月 31 日以前拥有的有效〈**希望交流俱乐部会员卡**〉可持续使用，按原会员章程执行，不进行新卡替换（直到不履行会员须知，旧卡被合理取消为止）。2001 年 1 月 1 日〈希望交流俱乐部〉将启用新会员卡，按俱乐部新章程执行。

此卡最终解释权在北京希望电子出版社。

请用正楷认真填写此表，以便我们准确记录您的信息，与您及时联系

《希望书盘交流俱乐部》会员回执表

姓　　名		年龄		职业	
工作单位				学历	
通讯地址			邮编		性别
			电话		
E-Mail			卡号		

北京希望电子出版社邮购部

好书可开启心灵窗户 工作事半功倍 卓越的品质与信誉是希望的座佑铭

高等院校计算机与软件教材系列



CX-83406
定价：39.00 元
ISBN:7900071652



CX-83476
定价：39.00 元
ISBN:7900071377



CX-83588
定价：30.00 元
ISBN:7900088113



CX-83327
定价:36.00 元
ISBN:7900056971



CX-83333
定价:38.00 元
ISBN:7900049398



CX-3138
定价:22.00 元
ISBN:7801441087



CX-82819
定价：28.00 元
ISBN:7900024387



CX-83200
定价：30.00 元
ISBN:7900049894



CX-83415
定价:25.00 元
ISBN:7900071768

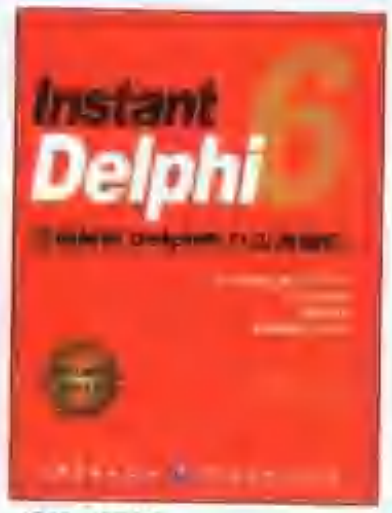
北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

地址：北京中关村 083 信箱北京希望电子出版社（邮编 100080）
电话：010-62562329，010-62633308 传真：010-62579874
E-mail: grh@hope.com.cn，更多的信息请访问 www.bhp.com.cn

专业编程人员成长之路



CX-3567
定价:40.00 元
ISBN:7801440838



CX-3568
定价:40.00 元
ISBN:7801440838



CX-3571
定价:40.00 元
ISBN:7801440838



CX-3572
定价:40.00 元
ISBN:7801440838



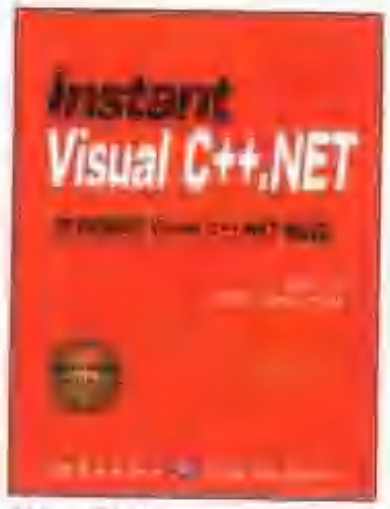
CX-3747
定价:39.00 元
ISBN:7900101012



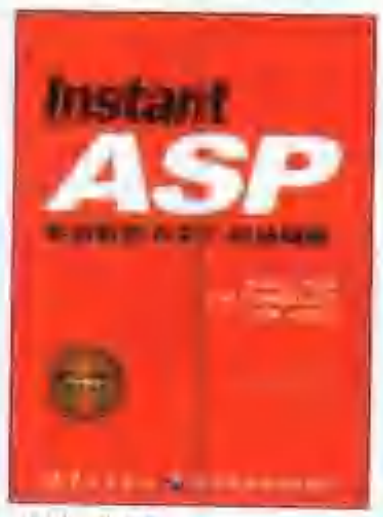
CX-3569
定价:40.00 元
ISBN:7801440838



CX-3581
定价:40.00 元
ISBN:7801440838



CX-3583
定价:40.00 元
ISBN:7801440838



CX-3591
定价:40.00 元
ISBN:7801440838



CX-3686
定价:42.00 元
ISBN:7900088717



CX-83576
定价: 58.00 元
ISBN:7980007700



CX-83598
定价: 30.00 元
ISBN:7900088040



CX-83597
定价: 55.00 元
ISBN:7900088024



CX-83599
定价: 35.00 元
ISBN:7900088075



CX-3660
定价:35.00 元
ISBN:7900088490



CX-83620
定价: 43.00 元
ISBN:7900088067



CX-83611
定价: 35.00 元
ISBN:7900088083



CX-83613
定价: 35.00 元
ISBN:7900088059



CX-83618
定价: 33.00 元
ISBN:7900088172



CX-83657
定价: 46.00 元
ISBN:7900088458

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

地址: 北京中关村 083 信箱北京希望电子出版社 (邮编 100080)
电话: 010-62562329, 010-62633308 传真: 010-62579874
E-mail: grh@hope.com.cn, 更多的信息请访问 www.bhp.com.cn